Geo1000 – Python Programming – Assignment 3

Due: Friday, October 11, 2019 (18h00)

Introduction

You are expected to create 2 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 provides a program that runs, does the correct things (as described in the assignment), your code is decent (e.g. use of proper variable names / indentation / etc) and submitted as required (e.g. on time).

It is due: **Friday, October 11, 2019 (18h00)**.

Note that if you submit your assignment after the deadline, 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, October 11, 2019, 18h15, the maximum amount of points that then can be obtained for the assignment is 90 $(100 - 10 = 90)$.

Submit the resulting program files (`patterns.py`, `loader.py`, `transformer.py`, `writer.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 that is handed in, starts with the following comment (augment `Authors` and `Studentnumbers` with your own names and numbers):

```
# GEO1000 - Assignment 3
# Authors:
# Studentnumbers:
```

1 A stack of squares – patterns.py (50 points)

Write a program to produce a text file with each of the following patterns with stacked squares (see Figure 2, 3 and 4). The ratio of the sizes of the squares (how much smaller every square is compared to a square on the level before) is set to 2.2.

The rectangle with p_1, p_2, p_3, p_4 from Figure 1 can be represented in Well Known Text (WKT) format as:

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

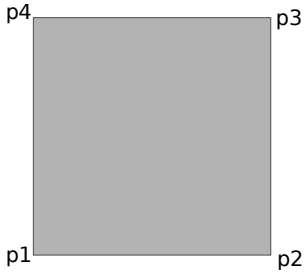


Figure 1: Square with coordinates

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' input option. Therefore the very first line of this text file should read `geometry`. The rest of the text file contains the well known text of the squares generated (every square on its own line). Do not forget to add an end of line character (`\n`) when you write to the text file.

Figure 2: Pattern A

Figure 3: Pattern B

Figure 4: Pattern C

The steps you need to take:

- Make a function that outputs the Well Known Text of a rectangle as Polygon (make sure that the coordinates are ordered counterclockwise, as in the example of Figure 1).
- 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.
- Implement 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.

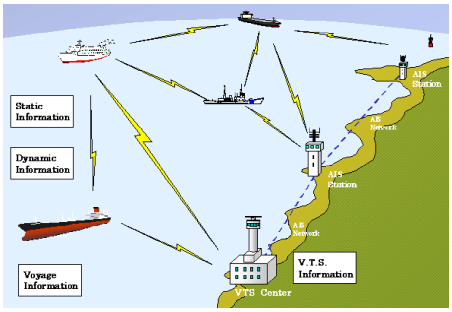
Start from the following skeleton. Do not use any imports.

[1] http://qgis.org/

```
# GEO1000 - Assignment 3
# Authors:
# Studentnumbers:


def wkt(p1, p2, p3, p4):
    """Returns Well Known Text string of
    square which is defined by 4 points.

    Arguments:

      p1--p4: 2-tuple of floats, the 4 corners of the square

    Returns:

      str (WKT of square) - POLYGON((x1 y1, x2 y2, x3 y3, x4 y4, x1 y1))

      note:
          - Order of coordinates (counterclockwise):
              p1 = bottom left corner,
              p2 = bottom right corner,
              p3 = top right corner,
              p4 = top left corner
          - Bottom left corner has to be repeated
          - Coordinates should be output with *6* digits behind the dot.
    """
    pass


def pattern_a(l, c, size, ratio, file_nm):
    """Draw a pattern of squares.

    Arguments:
        l - level to draw
        c - 2-tuple of floats (the coordinates of center of the square)
        size - half side length of the square
        ratio - how much smaller the next square will be drawn
        file_nm - file name of file to write to

    Returns:
        None
    """
    pass


def pattern_b(l, c, size, ratio, file_nm):
    """Draw a pattern of squares.

    Arguments:
        l - level to draw
        c - 2-tuple of floats (the coordinates of center of the square)
        size - half side length of the square
        ratio - how much smaller the next square will be drawn
        file_nm - file name of file to write to

    Returns:
        None
    """
    pass


def pattern_c(l, c, size, ratio, file_nm):
    """Draw a pattern of squares.

    Arguments:
        l - level to draw
        c - 2-tuple of floats (the coordinates of center of the square)
```

```
        size - half side length of the square
        ratio - how much smaller the next square will be drawn
        file_nm - file name of file to write to

    Returns:
        None
    """
    pass


def main(n=1, c=(0.0, 0.0), size=10.0, ratio=2.2):
    """The starting point of this program.
    Writes for every output file the first line and allows
    to influence how the resulting set of squares look.

    Arguments:
        n - levels of squares that are produced
        c - coordinate of center of the drawing
        size - half side length of the first square
        ratio - how much smaller a square on the next level will be drawn

    """
    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):
        # FIXME: finish this function
        pass


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

2 AIS Position Messages – loader.py, transformer.py, writer.py, ais.py, bitlist.py (50 points)



(a) AIS system overview [a].

(b) Man over board, carrying an AIS transponder [a].

[a] Image taken from: www6.kaiho.mlit.go.jp

[a] Image taken from: www.mcmurdomarine.com/ais-mob-devices

Figure 5: Automated Identification System (AIS) is used for improving safety at sea and at inland waters

AIS stands for Automated Identification System. The AIS system (Figure 5a) transmits regular information about a vessel (e.g. its position, speed, course, etc.) to other nearby vessels and vessel traffic management centres. For seagoing vessels and large inland vessels it is mandatory to carry an AIS transponder. Depending on the vessel its cruising speed or whether is anchored or moored, a transponder broadcasts its positions via VHF radio in intervals ranging from 2 seconds to 3 minutes. Every 6 minutes voyage related information is broadcast. Also other devices, e.g. man-over-board devices as shown in Figure 5b, can broadcast their position to others

Table 1: Characters in the payload and their corresponding 6 bits

Char	Bits	Char	Bits	Char	Bits	Char	Bits
0	000000	@	010000	P	100000	h	110000
1	000001	A	010001	Q	100001	i	110001
2	000010	B	010010	R	100010	j	110010
3	000011	C	010011	S	100011	k	110011
4	000100	D	010100	T	100100	l	110100
5	000101	E	010101	U	100101	m	110101
6	000110	F	010110	V	100110	n	110110
7	000111	G	010111	W	100111	o	110111
8	001000	H	011000	’	101000	p	111000
9	001001	I	011001	a	101001	q	111001
:	001010	J	011010	b	101010	r	111010
;	001011	K	011011	c	101011	s	111011
<	001100	L	011100	d	101100	t	111100
=	001101	M	011101	e	101101	u	111101
>	001110	N	011110	f	101110	v	111110
?	001111	O	011111	g	101111	w	111111

nearby.

For this programming assignment it is your task to make a program that transform an input file with raw AIS messages into a tab separated value text file, so that the information contained within the raw AIS messages can be read with QGIS (to visualize the position of the vessels).

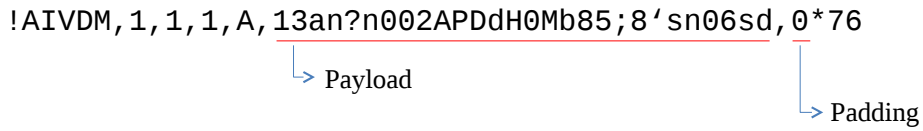


Figure 6: Structure of the raw AIS messages

You are given a text file with raw AIS position messages (i.e. AIS messages of type 1, 2 and 3). Figure 6 illustrates how the these messages in the text file are structured. For your program, you will need from each raw message its payload and padding (in the example the payload is the string: `13an?n002APDdH0Mb85;8’sn06sd` and the padding is the integer `0`). For each message, you also need the time when the message was received (stored in the file as UTC time and separated by a tab character from the raw AIS message).

Each character in the payload of the message encodes 6 bits of information by using an agreed upon encoding table (see Table 1). For the sample payload this leads to a list of bits as follows (only the first 4 characters and their corresponding bits are shown):

```
13an...
000001 000011 101001 110110 ...
```

Table 2 shows that by interpreting a subset of the bits of the message a field can be obtained. For the example, the bits 0 upto and including 5 store the `msgtype` field (which in this case is 1), the bits 6 upto and including 7 store the `repeat` field, which for the example is `0`, etc.

The message fields can be decoded by using an instance of the `BitList` class (that is also given to you). Note, that this class demonstrates the principle of information hiding, see § 17.11 in Think Python – as user of (a programmer working with) the BitList class you do not need to know how the bits are stored. After instantiating the class correctly (giving it a payload and padding value),

Table 2: Fields in AIS position messages (msgtype: 1, 2, 3)

Bits	Len	Name (key to be used)	Description	Signed / Unsigned	Post Processing Notes
0-5	6	msgtype	Message Type	u	
6-7	2	repeat	Repeat Indicator	u	
8-37	30	mmsi	MMSI	u	
38-41	4	status	Navigation Status	u	
42-49	8	turn	Rate of Turn (ROT)	s	
50-59	10	speed	Speed Over Ground (SOG)	u	divide by 10.0
60-60	1	accuracy	Position Accuracy	u	
61-88	28	lon	Longitude	s	geo (obtain WGS'84 coordinate)
89-115	27	lat	Latitude	s	geo (obtain WGS'84 coordinate)
116-127	12	course	Course Over Ground (COG)	u	divide by 10.0
128-136	9	heading	True Heading (HDG)	u	divide by 10.0
137-142	6	second	Second in UTC	u	
143-144	2	maneuver	Maneuver Indicator	u	
145-147	3	—	Spare	u	
148-148	1	raim	RAIM flag	u	
149-167	19	radio	Radio status	u	

you can use the `ubits` and `sbits` methods (to get signed or unsigned integers from the list of bits[2]). This enables you to get to the values of the fields.

The steps you need to make are as follows (see also the docstrings of the skeleton code provided):

1. Read the timestamp, payload and padding of each raw AIS message from the logfile into a list of tuples, where each tuple consist of `(timestamp: str, payload: str, padding: int)`.

   Functions to implement: `get_payload` and `read_payloads` (`loader.py`)

2. Transform this list of tuples with 3 elements into a list with tuples of 2 elements, where each tuple in the list has: `(timestamp: str, bit list instance of payload: Bitlist)`.

   Function to implement: `as_timestamp_bitlist` (`transformer.py`)

3. Decode all fields of each AIS position message by using the `ubits` and `sbits` methods on a BitList instance into a dictionary. The resulting dictionary should have the correct names as keys (as string, see Table 2 for the names of the fields to be used as dictionary key). Also, ensure that the timestamp is part of the decoded dictionary (the name for the key to use in this case: `timestamp`). The spare field (bits 145–147) of every message should be ignored and is not to be added to the dictionary.

   Note that some fields need some additional processing (see the column Post Processing Notes in Table 2). Post process the fields for which this is necessary (i.e. speed, lon, lat, course, heading) by updating the relevant items in the dictionary. For this, use the function `postprocess_msg_dict` (that calls the functions `div10` and `geo`) for performing the post processing.

   The dictionaries with decoded and formatted information are returned in a list.

   Functions to implement: `as_dicts`, `decode_msg_dict`, `postprocess_msg_dict` (`transformer.py`)

[2]This website gives more background about signed and unsigned integers: https://web.archive.org/web/20180111033110/http://kias.dyndns.org/comath/13.html

4. Make a function to generate a tab-separated values file, using the list of dictionaries. Make sure that the first line gives all names of the message fields (separated by tabs). Every other line in the file contains the contents of exactly one AIS position message of the input file. All fields of the message (including its timestamp) are separated by a tab character.

   The order in which the fields should be written to the file should be as follows:

   ```
   timestamp
   msgtype
   repeat
   mmsi
   status
   turn
   speed
   accuracy
   lon
   lat
   course
   heading
   second
   maneuver
   raim
   radio
   ```

   Function to implement: `write_tsv` (`writer.py`)

Once you have implemented all functions correctly, you should be able to run `ais.py` to perform the conversion. You should then be able to read the output file in QGIS with 'Add delimited text layer' (note that the coordinates are in the WGS'84 system, EPSG:4326).

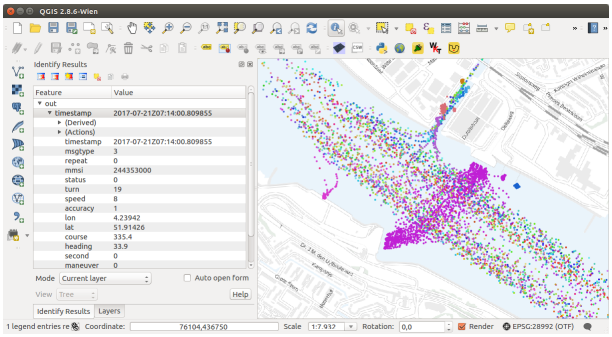


Figure 7: Decoded AIS data loaded in QGIS, showing the fields for a position message

Start from the following skeleton:

ais.py

```
from loader import read_payloads
from transformer import as_timestamp_bitlist, as_dicts
from writer import write_tsv

def main(in_filenm, out_filenm):
    """
    A program to transform a logfile with raw AIS messages into a
    tab-separated text file readable with QGIS
    """
    payloads = read_payloads(in_filenm)
    lst = as_timestamp_bitlist(payloads)
```

```
    lst = as_dicts(lst)
    write_tsv(lst, out_filenm)


if __name__ == "__main__":
    main("aislog.txt", "aislogout.txt")
```

loader.py

```
# GEO1000 - Assignment 3
# Authors:
# Studentnumbers:


def get_payload(raw_msg):
    """
    Returns tuple of payload and padding of a given raw AIS message

    For the raw AIS message:

        !AIVDM,1,1,1,A,13an?n002APDdH0Mb85;8'sn06sd,0*76

    the payload is:

        13an?n002APDdH0Mb85;8'sn06sd

    the padding (the digit before the *) is:

        0

    Returns:
        tuple (payload:str, padding:int)
    """
    pass


def read_payloads(filenm):
    """
    Reads the AIS messages (timestamp, payload and padding) from the file

    Arguments:
        :filenm: name of the file to be opened

    Uses:
        get_payload to get the payload and padding from each raw AIS message

    Returns:
        A list with tuples:
        [(timestamp:str, payload:str, padding:int), ...]
    """
    pass


def _test():
    # use this function to test your implementation
    pass


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

transformer.py

```
# GEO1000 - Assignment 3
# Authors:
# Studentnumbers:
```

```
from bitlist import BitList


def as_timestamp_bitlist(lst):
    """
    Transforms a list with:
      [(timestamp0, payload0, padding0), ..., (timestampn, payloadn, paddingn)]
    into a list with:
      [(timestamp0:str, bitlist0:BitList), ...]

    Returns:
        list with tuples
    """
    pass


def as_dicts(lst):
    """
    Transforms a list with:
      [(timestamp0:str, bitlist0:BitList), ...]
    into:
      [{'msgtype': 1, ...}, ...]

    Uses:
      decode_msg_dict, postprocess_msg_dict

    Returns:
        list with tuples
    """
    pass


def decode_msg_dict(timestamp, bitlist):
    """
    Decode a BitList instance to a dictionary

    Arguments:
        timestamp: str
        bitlist: BitList instance

    Returns:
        Dictionary with keys/values: timestamp and all fields
        for the position message

    **Note, values of the fields are all (signed or unsigned) integers!**
    """
    pass


def postprocess_msg_dict(msg):
    """
    Modifier function, post processes the fields:
        speed, lon, lat, course and heading

    Arguments:
        msg: dict (with all fields + timestamp of position message)

    Uses:
        functions: div10 and geo

    Returns:
        None
    """
    pass


def div10(field):
```

```
    """
    Divide a field by 10.0
    """
    return field / 10.


def geo(field):
    """
    Divide field by 600000.0 and rounds to 5
    """
    return round(field / 600000., 5)


def _test():
    # use this function to test your implementation
    pass


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

writer.py

```
# GEO1000 - Assignment 3
# Authors:
# Studentnumbers:

def write_tsv(lst, filenm_out):
    """
    Writes Tab Separated values to a file with name filenm_out

    Arguments:

        lst: list of dictionaries with the message content
             [{'msgtype': 1, ...}, {...}, ...]
        filenm_out: string specifying name of the file to use for output

    """
    pass


def _test():
    # use this function to test your implementation
    pass


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

Do not use any (additional) imports.