Lab 4

COMP9021, Session 2, 2018

1 A triangle of characters

Write a program characters_triangle.py that gets a strictly positive integer N as input and outputs a triangle of height N, following this kind of interaction:

```
$ python3 characters_triangle.py
Enter strictly positive number: 13
            A
           BCB
          DEFED
         GHIJIHG
        KLMNONMLK
       PQRSTUTSRQP
      VWXYZABAZYXWV
     CDEFGHIJIHGFEDC
    KLMNOPQRSRQPONMLK
   TUVWXYZABCBAZYXWVUT
  DEFGHIJKLMNMLKJIHGFED
 OPQRSTUVWXYZYXWVUTSRQPO
ABCDEFGHIJKLMLKJIHGFEDCBA
```

Two built-in functions are useful for this exercise:

- `ord()` returns the integer that encodes the character provided as argument;
- `chr()` returns the character encoded by the integer provided as argument.

For instance:

```
>>> ord('A')
65
>>> chr(65)
'A'
```

Consecutive uppercase letters are encoded by consecutive integers. For instance:

```
>>> ord('A'), ord('B'), ord('C')
(65, 66, 67)
```

2 Pascal triangle

Write a program pascal_triangle.py that prompts the user for a number N and prints out the first $N+1$ lines of Pascal triangle, making sure the numbers are nicely aligned, following this kind of interaction.

```
$ python3 pascal_triangle.py
Enter a nonnegative integer: 3
   1
  1 1
 1 2 1
1 3 3 1
$ python3 pascal_triangle.py
Enter a nonnegative integer: 7
               1
             1   1
           1   2   1
         1   3   3   1
       1   4   6   4   1
     1   5  10  10   5   1
   1   6  15  20  15   6   1
 1   7  21  35  35  21   7   1
$ python3 pascal_triangle.py
Enter a nonnegative integer: 11
                                   1
                                1     1
                             1     2     1
                          1     3     3     1
                       1     4     6     4     1
                    1     5    10    10     5     1
                 1     6    15    20    15     6     1
              1     7    21    35    35    21     7     1
           1     8    28    56    70    56    28     8     1
        1     9    36    84   126   126    84    36     9     1
     1    10    45   120   210   252   210   120    45    10     1
  1    11    55   165   330   462   462   330   165    55    11     1
```

3 Encoding pairs of integers as natural numbers (optional)

Write a program `plane_encoding.py` that implements a function `encode(a, b)` and a function `decode(n)` for the one-to-one mapping from the set of pairs of integers onto the set of natural numbers, that can be graphically described as follows:

16	15	14	13	12
17	4	3	2	11
18	5	0	1	10
19	6	7	8	9
20	21	...		

That is, starting from the point $(0, 0)$ of the plane, we move to $(1, 0)$ and then spiral counterclockwise:

- `encode(0,0)` returns `0` and `decode(0)` returns `(0,0)`
- `encode(1,0)` returns `1` and `decode(1)` returns `(1,0)`
- `encode(1,1)` returns `2` and `decode(2)` returns `(1,1)`
- `encode(0,1)` returns `3` and `decode(3)` returns `(0,1)`
- `encode(-1,1)` returns `4` and `decode(4)` returns `(-1,1)`
- `encode(-1,0)` returns `5` and `decode(5)` returns `(-1,0)`
- `encode(-1,-1)` returns `6` and `decode(6)` returns `(-1,-1)`
- `encode(0,-1)` returns `7` and `decode(7)` returns `(0,-1)`
- `encode(1,-1)` returns `8` and `decode(8)` returns `(1,-1)`
- `encode(2,-1)` returns `9` and `decode(9)` returns `(2,-1)`
- ...

4 Magic squares (Bachet, Siamese, and Lux methods are optional)

Write a program `magic_squares.py` that implements five functions: `print_square(square)`, `is_magic_square(square)`, `bachet_magic_square(n)`, `siamese_magic_square(n)`, and, finally, `lux_magic_square(n)`.

Given a positive integer n, a magic square of order n is a matrix of size $n \times n$ that stores all numbers from 1 up to n^2 and such that the sum of the n rows, the sum of the n columns, and the sum of the two diagonals is constant, hence equal to $n(n^2+1)/2$. The function `print_square(square)` prints a list of lists that represents a square, and the function `is_magic_square(square)` checks whether a list of lists is a magic square. For instance:

```
>>> from magic_squares import *
>>> print_square([[2,7,6], [9,5,1], [4,3,8]])
2 7 6
9 5 1
4 3 8
>>> is_magic_square([[2,7,6], [9,5,1], [4,3,8]])
True
>>> print_square([[2,7,6], [1,5,9], [4,3,8]])
2 7 6
1 5 9
4 3 8
>>> is_magic_square([[2,7,6], [1,5,9], [4,3,8]])
False
```

Given an odd positive integer n, the Bachet method produces a magic square of order n. Taking $n = 7$ as an example, this method

- starts with a square of size $2n - 1 \times 2n - 1$ filled as follows:

```
 .  .  .  .  .  .  1  .  .  .  .  .  .
 .  .  .  .  .  8  .  2  .  .  .  .  .
 .  .  .  . 15  .  9  .  3  .  .  .  .
 .  .  . 22  . 16  . 10  .  4  .  .  .
 .  . 29  . 23  . 17  . 11  .  5  .  .
 . 36  . 30  . 24  . 18  . 12  .  6  .
43  . 37  . 31  . 25  . 19  . 13  .  7
 . 44  . 38  . 32  . 26  . 20  . 14  .
 .  . 45  . 39  . 33  . 27  . 21  .  .
 .  .  . 46  . 40  . 34  . 28  .  .  .
 .  .  .  . 47  . 41  . 35  .  .  .  .
 .  .  .  .  . 48  . 42  .  .  .  .  .
 .  .  .  .  .  . 49  .  .  .  .  .  .
```

- then 4 times:
  - shift the n // 2 top rows n rows below:

```
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  . 22  . 16  . 10  .  4  .  .  .
  .  . 29  . 23  . 17  . 11  .  5  .  .
  . 36  . 30  . 24  . 18  . 12  .  6  .
 43  . 37  . 31  . 25  . 19  . 13  .  7
  . 44  . 38  . 32  1 26  . 20  . 14  .
  .  . 45  . 39  8 33  2 27  . 21  .  .
  .  .  . 46 15 40  9 34  3 28  .  .  .
  .  .  .  . 47  . 41  . 35  .  .  .  .
  .  .  .  .  . 48  . 42  .  .  .  .  .
  .  .  .  .  .  . 49  .  .  .  .  .  .
```

  - rotates clockwise by 90 degrees:

```
  .  .  .  .  .  . 43  .  .  .  .  .  .
  .  .  .  .  . 44  . 36  .  .  .  .  .
  .  .  .  . 45  . 37  . 29  .  .  .  .
  .  .  . 46  . 38  . 30  . 22  .  .  .
  .  . 47 15 39  . 31  . 23  .  .  .  .
  . 48  . 40  8 32  . 24  . 16  .  .  .
 49  . 41  9 33  1 25  . 17  .  .  .  .
  . 42  . 34  2 26  . 18  . 10  .  .  .
  .  . 35  3 27  . 19  . 11  .  .  .  .
  .  .  . 28  . 20  . 12  .  4  .  .  .
  .  .  .  . 21  . 13  .  5  .  .  .  .
  .  .  .  .  . 14  .  6  .  .  .  .  .
  .  .  .  .  .  .  7  .  .  .  .  .  .
```

Eventually, one reads the magic square off the centre:

```
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  . 22 47 16 41 10 35  4  .  .  .
  .  .  .  5 23 48 17 42 11 29  .  .  .
  .  .  . 30  6 24 49 18 36 12  .  .  .
  .  .  . 13 31  7 25 43 19 37  .  .  .
  .  .  . 38 14 32  1 26 44 20  .  .  .
  .  .  . 21 39  8 33  2 27 45  .  .  .
  .  .  . 46 15 40  9 34  3 28  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .
```

For instance:

```
>>> print_square(bachet_magic_square(7))
22 47 16 41 10 35  4
 5 23 48 17 42 11 29
30  6 24 49 18 36 12
13 31  7 25 43 19 37
38 14 32  1 26 44 20
21 39  8 33  2 27 45
46 15 40  9 34  3 28
```

Given an odd positive integer n, the Siamese method produces a magic square of order n. This method starts with 1 put at the centre of the first row, and having placed number $k < n^2$, places number $k + 1$ by moving diagonally up and right by one cell, wrapping around when needed (as if a torus was made out of the square), unless that cell is already occupied, in which case $k + 1$ is placed below the cell where k is (with no need to wrap around). For instance:

```
>>> print_square(siamese_magic_square(7))
30 39 48  1 10 19 28
38 47  7  9 18 27 29
46  6  8 17 26 35 37
 5 14 16 25 34 36 45
13 15 24 33 42 44  4
21 23 32 41 43  3 12
22 31 40 49  2 11 20
```

Given a positive integer n of the form $4 * k + 2$ with k a strictly positive integer, the LUX method produces a magic square of order n. This method proceeds as follows.

- Consider a matrix of size $2k + 1 \times 2k + 1$ that consists of:
  - $k + 1$ rows of Ls,
  - 1 row of Us, and
  - $k - 1$ rows of Xs,

  and then exchange the U in the middle with the L above it. For instance, when $n = 10$, that matrix is:

  ```
  L  L  L  L  L
  L  L  L  L  L
  L  L  U  L  L
  U  U  L  U  U
  X  X  X  X  X
  ```

- Explore all cells of this matrix as for the Siamese method, that is, starting at the cell at the centre of the first row, and then by moving diagonally up and right by one cell, wrapping around when needed (as if a torus was made out of the matrix), unless that cell has been visited already, in which case one moves down one cell (with no need to wrap around). The contents of every visited cell is then replaced by
  - if the cell contains L,

    ```
    i+4 i+1
    i+2 i+3
    ```

  - if the cell contains U,

    ```
    i+1 i+4
    i+2 i+3
    ```

  - if the cell contains X with i being the last number that has been used (starting with $i = 0$),

    ```
    i+1 i+4
    i+3 i+2
    ```

For instance:

```
>>> print_square(lux_magic_square(10))
 68  65  96  93   4   1  32  29  60  57
 66  67  94  95   2   3  30  31  58  59
 92  89  20  17  28  25  56  53  64  61
 90  91  18  19  26  27  54  55  62  63
 16  13  24  21  49  52  80  77  88  85
 14  15  22  23  50  51  78  79  86  87
 37  40  45  48  76  73  81  84   9  12
 38  39  46  47  74  75  82  83  10  11
 41  44  69  72  97 100   5   8  33  36
 43  42  71  70  99  98   7   6  35  34
```

5 Map of CO_2 emissions (optional, needs a module not installed on CSE computers)

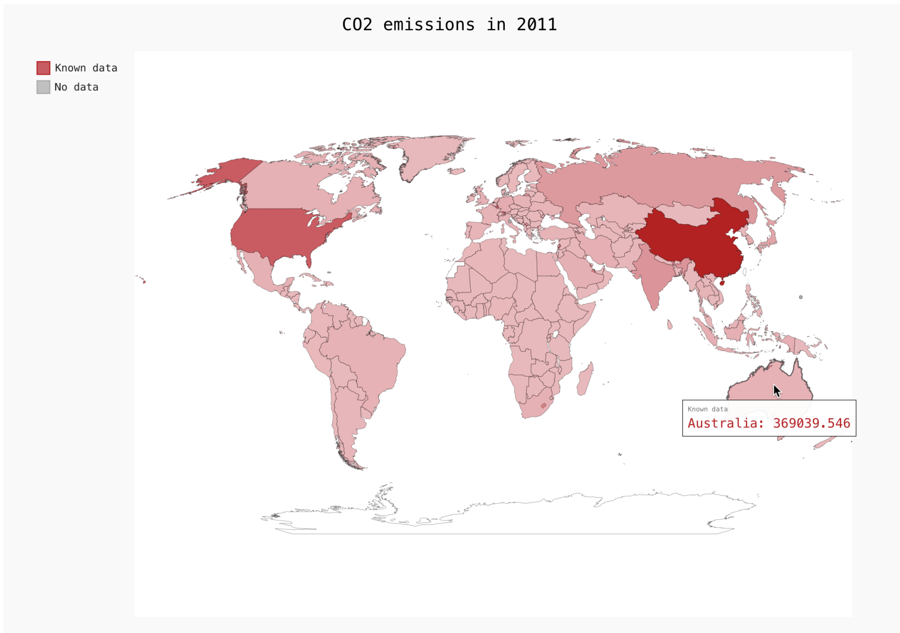
Write a program that extracts from the file `API_EN.ATM.CO2E.KT_DS2_en_csv_v2.csv`, stored in the subdirectory `API_EN` of the working directory, the country CO_2 emissions for the year 2011. Some data in this file are for entities different to countries, or for countries which are not values of the `COUNTRIES` dictionary of the `pygal.maps.world` module. The program will produce an output of the form

```
Leaving out Aruba
Leaving out Arab World
Leaving out American Samoa
Leaving out Antigua and Barbuda
Leaving out Bahamas, The
...
Leaving out Latin America & Caribbean (all income levels)
Leaving out Least developed countries: UN classification
Leaving out Low income
Leaving out Lower middle income
Leaving out Low & middle income
...
Leaving out Virgin Islands (U.S.)
Leaving out Vanuatu
Leaving out West Bank and Gaza
Leaving out World
Leaving out Samoa
```

to let the user know of all those entities and countries, which will be ignored. Some countries are described differently in the dictionary and in the file; these countries will not be ignored. The data will be shown interactively on a map, created as an object of class `World` of the `pygal.maps.world` module, that can be displayed in a browser by opening a file named `CO2_emissions.svg`—check out `render_to_file()`. To create the `World` object from a dictionary having as keys the keys of `COUNTRIES`, check out `add()`. The map should have—check out the `Style` class from the `pygal.style` module:

- as title for the map, `CO2 emissions in 2011`;
- one group of data with `Known data` as legend and with `#B22222` as colour, another group of data with `No data` as legend and with `#A9A9A9` as colour, both with a font size of 10pt;
- tooltips providing standard display for the first group, but with the amount of CO_2 emissions replaced by `?` for the second group, both with a font size of 8pt.

Here is the map with the cursor hovering over Australia, for which the CO_2 emissions are known.



Here is the map with the cursor hovering over Puerto Rico, for which the CO_2 emissions are not known.