Assignment 1

COMP9021, Session 1, 2016

1. General matters

1.1. **Aims.** The purpose of the assignment is to:

- let you design solutions to simple problems;
- let you implement these solutions in the form of short python programs;
- practice the use of arithmetic computations, tests, repetitions, lists, tuples, dictionnaries, deques, reading from files.

1.2. **Submission.** Your programs will be stored in a number of files, with one file per exercise, of the appropriate name. After you have developed and tested your programs, upload your files using WebCMS. Assignments can be submitted more than once: the last version is marked. Your assignment is due by April 10, 11:59pm.

1.3. **Assessment.** Each of the 4 exercises is worth 2.5 marks. For all exercises, the automarking script will let each of your programs run for 30 seconds. Still you should not take advantage of this and strive for a solution that gives an immediate output for "reasonable" inputs.

Late assignments will be penalised: the mark for a late submission will be the minimum of the awarded mark and 10 minus the number of full and partial days that have elapsed from the due date.

The outputs of your programs should be **exactly** as indicated.

1.4. **Reminder on plagiarism policy.** You are permitted, indeed encouraged, to discuss ways to solve the assignment with other people. Such discussions must be in terms of algorithms, not code. But you must implement the solution on your own. Submissions are routinely scanned for similarities that occur when students copy and modify other people's work, or work very closely together on a single implementation. Severe penalties apply.

2. Factorial base

Write a program, stored in a file named `factorial_base.py`, that performs the following task.

- Prompts the user to input a nonnegative integer. If the input is not a nonnegative integer, then the program outputs an error message and exits.
- Outputs the conversion of that number in "factorial base": a number written in factorial base is of the form $d_k \dots d_1$ with $d_i \leq i$ for all $1 \leq i \leq k$, and its decimal value is $d_k \times k! + \cdots + d_1 \times 1!$. For instance, the decimal value of 2301 read in factorial base is $2 \times 4! + 3 \times 3! + 0 \times 2! + 1 \times 1! = 67$.

Here is a sample run of the program.

```
$ python3 factorial_base.py
Input a nonnegative integer: 12a
Incorrect input, giving up...
$ python3 factorial_base.py
Input a nonnegative integer: 12 13
Incorrect input, giving up...
$ python3 factorial_base.py
Input a nonnegative integer: -3
Incorrect input, giving up...
$ python3 factorial_base.py
Input a nonnegative integer: 67
Decimal 67 reads as 2301 in factorial base.
$ python3 factorial_base.py
Input a nonnegative integer: 2301
Decimal 2301 reads as 310311 in factorial base.
$ python3 factorial_base.py
Input a nonnegative integer: 310311
Decimal 310311 reads as 75354211 in factorial base.
```

3. Rubik's rectangle

Write a program, stored in a file named `rubiks_rectangle.py`, that performs the following task.

- Prompts the user to input the digits 1 to 8 (with possibly whitespace inserted anywhere), in some order, say $d_1\ d_2\ d_3\ d_4\ d_5\ d_6\ d_7\ d_8$, without repetition; if the input is incorrect, then the program outputs an error message and exits.
- Finds the minimal number of steps needed to go from the initial state, represented as

1	2	3	4
8	7	6	5

to the final state, represented as

d_1	d_2	d_3	d_4
d_8	d_7	d_6	d_5

following a sequence of transformations named *row exchange*, *right circular shift* and *middle clockwise rotation*, that transform a configuration of the form

k_1	k_2	k_3	k_4
k_8	k_7	k_6	k_5

into
  - for row exchange:

k_8	k_7	k_6	k_5
k_1	k_2	k_3	k_4

  - for right circular shift:

k_4	k_1	k_2	k_3
k_5	k_8	k_7	k_6

  - for middle clockwise rotation:

k_1	k_7	k_2	k_4
k_8	k_6	k_3	k_5

For instance, if the final configuration is 26845731 then 7 steps are sufficient (and necessary):
  - Initial configuration

1	2	3	4
8	7	6	5

  - followed by right circular shift

4	1	2	3
5	8	7	6

  - followed by middle clockwise rotation

4	8	1	3
5	7	2	6

  - followed by row exchange

5	7	2	6
4	8	1	3

  - followed by right circular shift

6	5	7	2
3	4	8	1

  - followed by middle clockwise rotation

6	4	5	2
3	8	7	1

  - followed by middle clockwise rotation

6	8	4	2
3	7	5	1

  - followed by right circular shift

2	6	8	4
1	3	7	5

which is the final configuration.

- Outputs the number of steps needed to reach the final configuration (this is always possible), using a message whose grammatical form depends on whether the number of steps is 0 or 1, or at least 2.

Here is a sample run of the program.

```
$ python3 rubiks_rectangle.py
Input final configuration: 01234567
Incorrect configuration, giving up...
$ python3 rubiks_rectangle.py
Input final configuration: 12345678
0 step is needed to reach the final configuration.
$ python3 rubiks_rectangle.py
Input final configuration: 2 6 8 4 5 7 3 1
7 steps are needed to reach the final configuration.
$ python3 rubiks_rectangle.py
Input final configuration: 7215  4368
16 steps are needed to reach the final configuration.
$ python3 rubiks_rectangle.py
Input final configuration: 1   5   3   2   4   6   7   8
18 steps are needed to reach the final configuration.
```

Hint: Use `collections.deque`

4. A word game

Write a program, stored in a file named `highest_scoring_words.py`, that performs the following task.

- Prompts the user to input between 3 and 10 lowercase letters (with possibly whitespace inserted anywhere); if the input contains too few or too many lowercase letters or any character which is neither a lowercase letter nor whitespace, then the program outputs an error message and exits.
- Finds in the file `wordsEn.txt`, assumed to be stored in the working directory, the words built from the letters input by the user (with the exclusion of any other character) with highest score, if any; the score of a word is defined as the sum of the values of the letters that make up that word, the value of each letter being defined as follows:

a 2	b 5	c 4	d 4	e 1	f 6
g 5	h 5	i 1	j 7	k 6	l 3
m 5	n 2	o 3	p 5	q 7	r 2
s 1	t 2	u 4	v 6	w 6	x 7
y 5	z 7				

- Outputs a specific message if there is no such word; otherwise, outputs the highest score and all words with that score, one word per line, with a different introductory message depending on whether there is a unique such word (in which case the introductory message is on the same line as the word) or at least two such words (in which case the introductory message is on a line of its own and all words are preceded with 4 spaces).

Here is a sample run of the program.

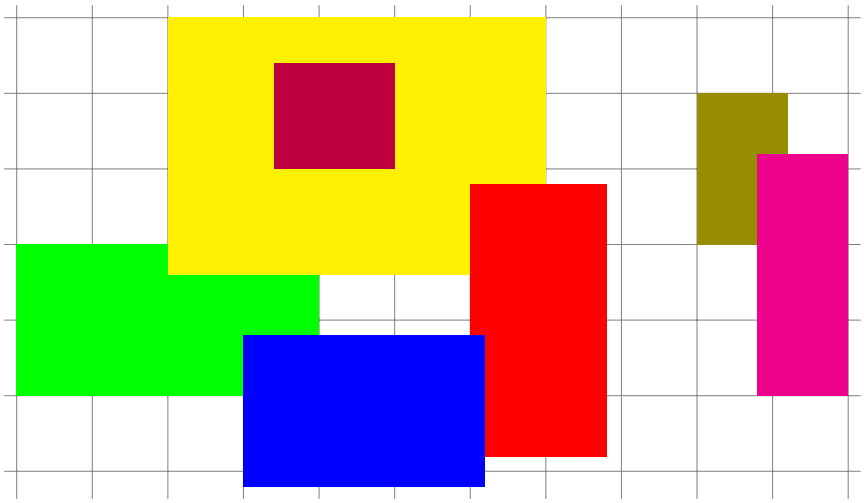
```
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: abc2ef
Incorrect input, giving up...
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: ab
Incorrect input, giving up...
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: abcdefghijk
Incorrect input, giving up...
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: zz zz zz
No word is built from some of those letters.
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: a  a   a
The highest score is 2.
The highest scoring word is a
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: a e i o u
The highest score is 8.
The highest scoring words are, in alphabetical order:
     iou
     oui
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: prmgroa
The highest score is 24.
The highest scoring word is program
```

```
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: a b  e    o    r   a t
The highest score is 16.
The highest scoring word is abator
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: r a m m o x y
The highest score is 17.
The highest scoring words are, in alphabetical order:
     mayor
     moray
     moxa
     oryx
$ python3 highest_scoring_words.py
Enter between 3 and 10 lowercase letters: eaeo rtsmn
The highest score is 17.
The highest scoring words are, in alphabetical order:
     matrons
     transom
```

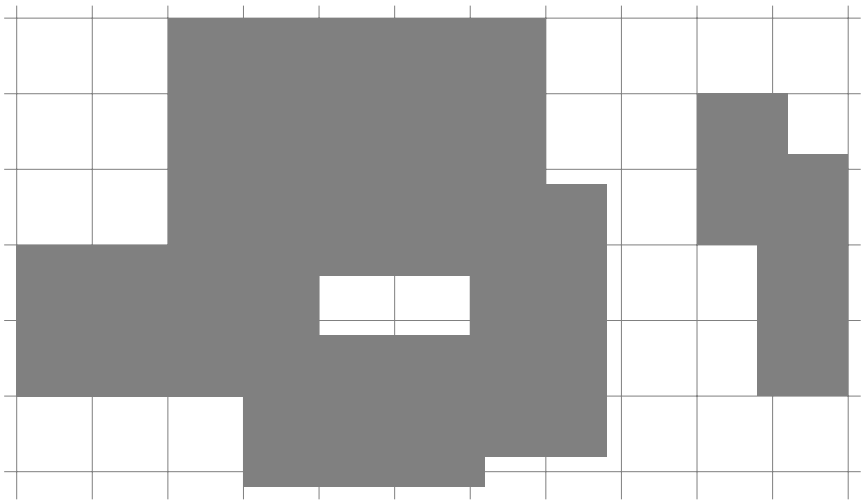
5. Overlapping photographs

Write a program, stored in a file named `perimeter.py`, that performs the following task.

- Prompts the user to input the name of text file assumed to be stored in the working directory. We assume that if the name is valid then the file consists of lines all containing 4 integers separated by whitespace, of the form x_1 y_1 x_2 y_2 where (x_1, y_1) and (x_2, y_2) are meant to represent the coordinates of two opposite corners of a rectangle. With the provided file `frames_1.txt`, the rectangles can be represented as follows, using from top bottom the colours green, yellow, red, blue, purple, olive, and magenta.



  We assume that all rectangles are distinct and either properly overlap or are disjoint (they do not touch each other by some of their sides or some of their corners).
- Computes and outputs the perimeter of the boundary, so with the sample file `perimeter.py`, the sum of the lengths of the (external or internal) sides of the following picture.



Here is a sample run of the program with the two provided sample files.

```
$ python3 perimeter.py
Which data file do you want to use? frames_1.txt
The perimeter is: 228
$ python3 perimeter.py
Which data file do you want to use? frames_2.txt
The perimeter is: 9090
```