Assignment 1

COMP9021, Session 2, 2017

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 Ed. Assignments can be submitted more than once: the last version is marked. Your assignment is due by September 3, 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. A pivoting die

Consider the board below, which can be imagined as being infinite, so only a small part is represented. A die is placed on cell number 1 of the board in the position indicated: 3 at the top, 2 at the front (towards cell number 4 of the board), and 1 on the right (towards cell number 2 of the board). Also, 4 is opposite to 3, 5 is opposite to 2, and 6 is opposite to 1. The die can be moved from cell 1 to cell 2, then to cell 3, then to cell 4..., each time pivoting by 90 degrees in the appropriate direction (to the right, forwards, to the left, or backwards). For instance, after it has been moved from cell 1 to cell 2, 2 is still at the front but 6 is at the top and 3 is on the right.

43	44	45	46	47	48	49	50
42	21	22	23	24	25	26	
41	20	7	8	9	10	27	
40	19	6	1	2	11	28	
39	18	5	4	3	12	29	
38	17	16	15	14	13	30	
37	36	35	34	33	32	31	

Write a python program, named pivoting_die.py, that prompts the user for a natural number N at least equal to 1, and outputs the numbers at the top, the front and the right after the die has been moved to cell N.

Here is a possible interaction.

```
$ python3 pivoting_die.py
Enter the desired goal cell number: A
Incorrect value, try again
Enter the desired goal cell number:
Incorrect value, try again
Enter the desired goal cell number: -1
Incorrect value, try again
Enter the desired goal cell number: 0
Incorrect value, try again
Enter the desired goal cell number: 1
On cell 1, 3 is at the top, 2 at the front, and 1 on the right.
$ python3 pivoting_die.py
Enter the desired goal cell number: 29
On cell 29, 3 is at the top, 2 at the front, and 1 on the right.
$ python3 pivoting_die.py
Enter the desired goal cell number: 2006
On cell 2006, 4 is at the top, 1 at the front, and 2 on the right.
```

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.
```

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. Poker dice

Write a program named `poker_dice.py` that simulates the roll of 5 dice, at most three times, as described at `http://en.wikipedia.org/wiki/Poker_dice` as well as a given number of rolls of the 5 dice to evaluate the probabilities of the various hands.

The next three pages show a possible interaction.

The following observations are in order.

- Your program has to define the functions `play()` and `simulate()`, but of course it will likely include other functions.
- The hands are displayed in the order Ace, King, Queen, Jack, 10 and 9.
- All dice can be kept by inputting either `all`, or `All`, or the current hand in any order.
- No die is kept by inputting nothing.
- We have control over what is randomly generated by using the `seed()` function at the prompt, but do not make use of `seed()` in the program.
- To roll a die, we use `randint(0, 5)` **with 0, 1, 2, 3, 4 and 5 corresponding to Ace, King, Queen, Jack, 10 and 9, respectively**.

:

```
$ python3
Python 3.6.2 ...
>>> from random import seed
>>> import poker_dice
>>> seed(0)
>>> poker_dice.play()
The roll is: Ace Queen Jack Jack 10
It is a One pair
Which dice do you want to keep for the second roll? all
Ok, done.
>>> poker_dice.play()
The roll is: Queen Queen Jack Jack Jack
It is a Full house
Which dice do you want to keep for the second roll?  All
Ok, done.
>>> poker_dice.play()
The roll is: King King Queen 10 10
It is a Two pair
Which dice do you want to keep for the second roll?  King 10 Queen King 10
Ok, done.
>>> poker_dice.play()
The roll is: Ace King Queen 10 10
It is a One pair
Which dice do you want to keep for the second roll? 10 11
That is not possible, try again!
Which dice do you want to keep for the second roll? ace
That is not possible, try again!
Which dice do you want to keep for the second roll? 10 10
The roll is: King 10 10 10 9
It is a Three of a kind
Which dice do you want to keep for the third roll? all
Ok, done.
```

```
>>> poker_dice.play()
The roll is: Ace Ace Queen 9 9
It is a Two pair
Which dice do you want to keep for the second roll? Ace
The roll is: Ace Ace Queen Jack 10
It is a One pair
Which dice do you want to keep for the third roll? Ace
The roll is: Ace Queen Queen Jack 10
It is a One pair
>>> seed(2)
>>> poker_dice.play()
The roll is: Ace Ace Ace King Queen
It is a Three of a kind
Which dice do you want to keep for the second roll? Ace Ace Ace
The roll is: Ace Ace Ace 9 9
It is a Full house
Which dice do you want to keep for the third roll? all
Ok, done.
>>> poker_dice.play()
The roll is: King Queen Queen 10 10
It is a Two pair
Which dice do you want to keep for the second roll?
The roll is: Ace King Jack 10 9
It is a Bust
Which dice do you want to keep for the third roll?
The roll is: Queen Jack 10 9 9
It is a One pair
>>> seed(10)
>>> poker_dice.play()
The roll is: Ace Jack Jack 10 10
It is a Two pair
Which dice do you want to keep for the second roll? Jack 10 Jack 10
The roll is: Ace Jack Jack 10 10
It is a Two pair
Which dice do you want to keep for the third roll? Jack 10 Jack 10
The roll is: King Jack Jack 10 10
It is a Two pair
>>> seed(20)
>>> poker_dice.play()
The roll is: King Queen 9 9 9
It is a Three of a kind
Which dice do you want to keep for the second roll? 9 King 9 9
The roll is: King 9 9 9 9
It is a Four of a kind
Which dice do you want to keep for the third roll? 9 9 9 9
The roll is: Ace 9 9 9 9
It is a Four of a kind
```

```
>>> seed(0)
>>> poker_dice.simulate(10)
Five of a kind : 0.00%
Four of a kind : 0.00%
Full house     : 10.00%
Straight       : 0.00%
Three of a kind: 0.00%
Two pair       : 20.00%
One pair       : 60.00%
>>> poker_dice.simulate(100)
Five of a kind : 0.00%
Four of a kind : 0.00%
Full house     : 3.00%
Straight       : 4.00%
Three of a kind: 14.00%
Two pair       : 28.00%
One pair       : 45.00%
>>> poker_dice.simulate(1000)
Five of a kind : 0.10%
Four of a kind : 2.50%
Full house     : 3.80%
Straight       : 3.40%
Three of a kind: 17.20%
Two pair       : 20.60%
One pair       : 46.30%
>>> poker_dice.simulate(10000)
Five of a kind : 0.08%
Four of a kind : 1.99%
Full house     : 3.93%
Straight       : 3.33%
Three of a kind: 14.88%
Two pair       : 23.02%
One pair       : 46.58%
>>> poker_dice.simulate(100000)
Five of a kind : 0.08%
Four of a kind : 1.94%
Full house     : 3.85%
Straight       : 3.17%
Three of a kind: 15.47%
Two pair       : 23.02%
One pair       : 46.31%
```