Lab 5

COMP9021, Session 2, 2015

1 ☞ Magic squares

A 3×3 square whose cells contain every digit in the range 1–9 is said to be *magic* if the sums of the rows, the sums of the columns and the sums of the diagonals are all equal numbers.

Write a program that generates all magic squares.

Here is the beginning of a possible run of the program (“possible”, as the order of the output solutions can vary):

```
$ python3 question_1.py
  4  9  2
  3  5  7
  8  1  6

  6  7  2
  1  5  9
  8  3  4

  ...
```

2 ☞ Extracting information from a web page

Write a program that extracts titles from a front page of the Sydney Morning Herald, provided under the name SMH.txt, meant to be saved in the working directory. You are provided with the expected output, saved in the file `question_2_outputs.txt`, though you might do a better job and remove some of the titles (for instance, *The Lady who lives on the Moon* could go...). Make sure that the output does not include any unwanted HTML entity.

For this question, you probably need to use regular expressions.

3 A calendar program

Write a program that provides a variant on the Unix `cal` utility (in particular because it lets the weeks start on Monday, not Sunday), following this kind of interaction:

```
$ python3 calendar.py
I will display a calendar, either for a year or for a month in a year.
The earliest year should be 1753.
For the month, input at least the first three letters of the month's name.
Input year, or year and month, or month and year: 3194
                               3194

       January               February               March
 Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                 1  2      1  2  3  4  5  6      1  2  3  4  5  6
  3  4  5  6  7  8  9   7  8  9 10 11 12 13   7  8  9 10 11 12 13
 10 11 12 13 14 15 16  14 15 16 17 18 19 20  14 15 16 17 18 19 20
 17 18 19 20 21 22 23  21 22 23 24 25 26 27  21 22 23 24 25 26 27
 24 25 26 27 28 29 30  28                    28 29 30 31
 31
        April                  May                   June
 Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
              1  2  3                     1         1  2  3  4  5
  4  5  6  7  8  9 10   2  3  4  5  6  7  8   6  7  8  9 10 11 12
 11 12 13 14 15 16 17   9 10 11 12 13 14 15  13 14 15 16 17 18 19
 18 19 20 21 22 23 24  16 17 18 19 20 21 22  20 21 22 23 24 25 26
 25 26 27 28 29 30     23 24 25 26 27 28 29  27 28 29 30
                       30 31
         July                 August              September
 Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
              1  2  3   1  2  3  4  5  6  7            1  2  3  4
  4  5  6  7  8  9 10   8  9 10 11 12 13 14   5  6  7  8  9 10 11
 11 12 13 14 15 16 17  15 16 17 18 19 20 21  12 13 14 15 16 17 18
 18 19 20 21 22 23 24  22 23 24 25 26 27 28  19 20 21 22 23 24 25
 25 26 27 28 29 30 31  29 30 31              26 27 28 29 30
       October               November              December
 Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                 1  2      1  2  3  4  5  6            1  2  3  4
  3  4  5  6  7  8  9   7  8  9 10 11 12 13   5  6  7  8  9 10 11
 10 11 12 13 14 15 16  14 15 16 17 18 19 20  12 13 14 15 16 17 18
 17 18 19 20 21 22 23  21 22 23 24 25 26 27  19 20 21 22 23 24 25
 24 25 26 27 28 29 30  28 29 30              26 27 28 29 30 31
 31
```

In doing this exercise, you will have to find out (or just remember. . .) how leap years are determined, and what is so special about the year 1753. . .

```
$ python3 calendar.py
I will display a calendar, either for a year or for a month in a year.
The earliest year should be 1753.
For the month, input at least the first three letters of the month’s name.
Input year, or year and month, or month and year: 3194 Sept
    September 3194
 Mo Tu We Th Fr Sa Su
              1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
 26 27 28 29 30
$ python3 calendar.py
I will display a calendar, either for a year or for a month in a year.
The earliest year should be 1753.
For the month, input at least the first three letters of the month’s name.
Input year, or year and month, or month and year: dEcEm 3194
    December 3194
 Mo Tu We Th Fr Sa Su
              1  2  3  4
  5  6  7  8  9 10 11
 12 13 14 15 16 17 18
 19 20 21 22 23 24 25
 26 27 28 29 30 31
```

Lab 6

COMP9021, Session 2, 2015

1 ☞ Obtaining a sum from a subsequence of digits

Write a program that prompts the user for two numbers, say `available_digits` and `desired_sum`, and outputs the number of ways of selecting digits from `available_digits` that sum up to `desired_sum`. For instance, if `available_digits` is `12234` and `sum` is `5` then there are 4 solutions:

- one solution is obtained by selecting 1 and both occurrences of 2 ($1 + 2 + 2 = 5$);
- one solution is obtained by selecting 1 and 4 ($1 + 4 = 5$);
- one solution is obtained by selecting the first occurrence of 2 and 3 ($2 + 3 = 5$);
- one solution is obtained by selecting the second occurrence of 2 and 3 ($2 + 3 = 5$).

Here is a possible interaction:

```
$ python question_1.py
Input a number that we will use as available digits: 12234
Input a number that represents the desired sum: 5
There are 4 solutions.
$ python question_1.py
Input a number that we will use as available digits: 11111
Input a number that represents the desired sum: 5
There is a unique solution
$ python question_1.py
Input a number that we will use as available digits: 11111
Input a number that represents the desired sum: 6
There is no solution.
$ python question_1.py
Input a number that we will use as available digits: 1234321
Input a number that represents the desired sum: 5
There are 10 solutions.
```

2 ☞ Merging two strings into a third one

Say that two strings s_1 and s_2 can be merged into a third string s_3 if s_3 is obtained from s_1 by inserting arbitrarily in s_1 the characters in s_2, respecting their order. For instance, the two strings *ab* and *cd* can be merged into *abcd*, or *cabd*, or *cdab*, or *acbd*, or *acdb*, ..., but not into *adbc* nor into *cbda*.

Write a program that prompts the user for 3 strings and displays the output as follows:

- If no string can be obtained from the other two by merging, then the program outputs that there is no solution.
- Otherwise, the program outputs which of the strings can be obtained from the other two by merging.

Here is a possible interaction:

```
$ python question_2.py
Please input the first string: ab
Please input the second string: cd
Please input the third string: abcd
The third string can be obtained by merging the other two.
$ python question_2.py
Please input the first string: ab
Please input the second string: cdab
Please input the third string: cd
The second string can be obtained by merging the other two.
$ python question_2.py
Please input the first string: abcd
Please input the second string: cd
Please input the third string: ab
The first string can be obtained by merging the other two.
$ python question_2.py
Please input the first string: ab
Please input the second string: cd
Please input the third string: adcb
No solution
$ python question_2.py
Please input the first string: aaaaa
Please input the second string: a
Please input the third string: aaaa
The first string can be obtained by merging the other two.
$ python question_2.py
Please input the first string: aaab
Please input the second string: abcab
Please input the third string: aaabcaabb
The third string can be obtained by merging the other two.
$ python question_2.py
Please input the first string: ??got
Please input the second string: ?it?go#t##
Please input the third string: it###
The second string can be obtained by merging the other two.
```

Lab 7

COMP9021, Session 2, 2015

1 ☞ Change-making problem: greedy solution

Write a program that prompts the user for an amount, and outputs the minimal number of banknotes needed to yield that amount, as well as the detail of how many banknotes of each type value are used. The available banknotes have a face value which is one of \$1, \$2, \$5, \$10, \$20, \$50, and \$100.

Here are examples of interactions:

```
$ python question_1.py
Input the desired amount: 10

1 banknote is needed.
The detail is:
 $10: 1
$ python question_1.py
Input the desired amount: 739

12 banknotes are needed
The detail is:
$100: 7
 $20: 1
 $10: 1
  $5: 1
  $2: 2
$ python question_1.py
Input the desired amount: 35642

359 banknotes are needed
The detail is:
$100: 356
 $20: 2
  $2: 1
```

The natural solution implements a *greedy* approach: we always look for the largest possible face value to deduct from what remains of the amount.

Suppose that the available banknotes had a face value which was one of \$1, \$20, and \$50. For an amount of \$60, the greedy algorithm would not work, as it would yield one \$50 banknote and ten \$1 banknotes, so eleven banknotes all together, whereas we only need three \$20 banknotes.

2 ☞ Change-making problem: general solution

Write a program that prompts the user for the face values of banknotes and their associated quantities as well as for an amount, and if possible, outputs the minimal number of banknotes needed to match that amount, as well as the detail of how many banknotes of each type value are used.

Note: this question is flagged but in case it is used in the exam, only the minimal number of banknotes needed to match the amount would have to be computed, not the various solutions.

Here are examples of interactions:

```
$ python question_2.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

2 : 100
50: 100

Input the desired amount: 99

There is no solution.
$ python question_2.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

1  : 30
20 : 30
50 : 30

Input the desired amount: 60
There is a unique solution:
$20: 3
```

```
$ python question_2.py
Input pairs of the form 'value : number'
   to indicate that you have 'number' many banknotes of face value 'value'.
Input these pairs one per line, with a blank line to indicate end of input.

1: 100
2: 5
3: 4
10:5
20:4
30:1

Input the desired amount: 107

There are 2 solutions:
 $1: 1
 $3: 2
$10: 1
$20: 3
$30: 1

 $2: 2
 $3: 1
$10: 1
$20: 3
$30: 1
```

The natural approach makes use of the linear programming technique exemplified in the computation of the Levenshtein distance between two words.

3 ☞ A class to work exactly with fractions

Write a program that allows one to create fractions and work precisely with them, following this kind of interaction:

```
$ python
...
>>> from question_3 import Fraction
>>> f = Fraction()
Provide exactly two arguments.
>>> f = Fraction(1, 5, 12)
Provide exactly two arguments.
>>> f = Fraction(1, 'hi')
Provide an integer and a nonzero integer as arguments.
```

```
>>> f = Fraction(3, 0)
Provide an integer and a nonzero integer as arguments.
>>> f = Fraction(0, 3)
>>> f
Fraction(numerator = 0, denominator = 1)
>>> print(f)
0 / 1
>>> f = Fraction(120, -30)
>>> f
Fraction(numerator = -4, denominator = 1)
>>> print(f)
-4 / 1
>>> f = Fraction(-310532200, -46077863832)
>>> f
Fraction(numerator = 425, denominator = 63063)
>>> print(f)
425 / 63063
>>> f1 = Fraction(123, 456)
>>> f2 = Fraction(2345, 6789)
>>> f3 = f1 - f2
>>> print(f3)
-78091 / 1031928
>>> f3 = f1 * f2
>>> print(f3)
96145 / 1031928
>>> f1 / Fraction(0, 35)
Cannot divide by 0.
>>> f1 < f2
True
>>> f1 <= f2
True
>>> f1 > f2
False
>>> f1 >= f2
False
>>> f1 == f2
False
>>> f1 != f2
True
```

To find out whether the correct number of arguments has been provided, pass an argument of the form `*args` to `__init__` (the name `args` is arbitrary; what matters is the leading star, which makes `args` a tuple consisting of all positional arguments).

The implementation makes use of the following special methods (find out about them): `__repr__`, `__str__`, `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`.

Lab 8

COMP9021, Session 2, 2015

1 ☞ Doubly linked lists

Modify the module `linked_list.py` which is part of the material of the 8th lecture into a module `doubly_linked_list.py`, to process lists consisting of nodes with a reference to both next and previous nodes, so with the class `Node` defined as follows.

```
class Node:
    def __init__(self, value = None):
        self.value = value
        self.next_node = None
        self.previous_node = None
```

2 Using linked lists to represent polynomials

Write a program that implements a class `Polynomial`. An object of this class is built from a string that represents a polynomial, that is, a sum or difference of monomials.

- The leading monomial can be either an integer, or an integer followed by `x`, or an integer followed by `x^` followed by a nonnegative integer.
- The other monomials can be either a nonnegative integer, or an integer followed by `x`, or an integer followed by `x^` followed by a nonnegative integer.

Spaces can be inserted anywhere in the string.

A monomial is defined by the following class:

```
class Monomial:
    def __init__(self, coefficient = 0, degree = 0):
        self.coefficient = coefficient
        self.degree = degree
        self.next_monomial = None
```

A polynomial is a linked list of monomials, ordered from those of higher degree to those of lower degree. An implementation of the `__str__()` method allows one to print out a polynomial.

Next is a possible interaction.

```
>>> print(poly_13 / poly_6)
2x^5 - 93x^4 - 71x^3 + 8x^2 - 6x + 192
>>> print(poly_13 / poly_7)
-x^47 + x^23 - 12x^8 + 45x^6 + 6x^3 - x + 2
```

2 ☞ Using a stack to evaluate fully parenthesised expressions

Modify the program `postfix.py` from the 9th lecture so that a stack is used to evaluate an arithmetic expression written in infix, fully parenthesised, and built from natural numbers using the binary +, -, * and / operators. *Fully parenthesised* means that all expressions of the form e + e', e - e', e * e' and e / e' are surrounded by a pair of parentheses, brackets or braces. Of course a simple solution would be to replace all brackets and braces by parentheses and call `eval()`, but here we want to use a stack.

Hint: think of popping when and only when a closing parenthesis, bracket or brace is being processed.

Here is a possible interaction:

```
$ python
...
>>> from exercise_2 import *
>>> expression = FullyParenthesisedExpression('2')
>>> expression.evaluate()
2
>>> expression = FullyParenthesisedExpression('(2 + 3)')
>>> expression.evaluate()
5
>>> expression = FullyParenthesisedExpression('[(2 + 3) / 10]')
>>> expression.evaluate()
0.5
>>> expression = FullyParenthesisedExpression('(12 + [{[13 + (4 + 5)] - 10} / (7 * 8)])')
>>> expression.evaluate()
12.214285714285714
```

3 Back to context free grammars

A *context free* grammar is a set of *production rules* of the form

```
symbol_0 --> symbol_1 ...  symbol_n
```

where `symbol_0`, ..., `symbol_n` are either *terminal* or *nonterminal symbols*, with `symbol_0` being necessarily nonterminal. A symbol is a nonterminal symbol iff it is denoted by a word built from underscores or uppercase letters. A special nonterminal symbol is called the *start symbol*. The language *generated* by the grammar is the set of sequences of terminal symbols obtained by replacing

Lab 10

COMP9021, Session 2, 2015

1 ☞ Building a general tree

Consider a file named `tree.txt` containing numbers organised as a tree, a number at a depth of N in the tree being preceded with N tabs in the file. The file can also contain any number of lines with nothing but blank lines. Using the module `general_tree.py`, write a program that reads the contents of the file. If the file does not contain a proper representation of a tree then the program outputs an error message; otherwise, it builds the tree (an instance of GeneralTree()) and prints it out using the same representation as in the file (except for the possible blank lines of course). Here is a possible interaction:

```
$ python
...
$ cat tree.txt

2
        3
                1
        4




                5
                        7
                                8
                9

                        10
                        11
                        12



        6
1
$ python exercise_1.py
tree.txt does not contain the correct representation of a tree.
```

```
$ cat tree.txt

2
        3
                1
        4




                5
                                7
                                8
                9

        6
$ python exercise_1.py
tree.txt does not contain the correct representation of a tree.
$ cat tree.txt

2
        3
                1
        4




                5
                        7
                                8
                9

                        10
                        11
                        12



        6
$ python exercise_1.py
2
        3
                1
        4
                5
                        7
                                8
                9
                        10
                        11
                        12
        6
$
```

Lab 11

COMP9021, Session 2, 2015

1 ☞ Median and priority queues

Implement a class `Median` that allows one to manage a list L of values with the following operations:

- add a value in logarithmic time complexity;
- return the median in constant time complexity.

One possible design is to use two priority queues: a max priority queue to store the half of the smallest elements, and a min priority queue to store the half of the largest elements. Both priority queues have the same number of elements if the number of elements in L is even, in which case the median is the average of the elements at the top of both priority queues. Otherwise, one priority queue has one more element than the other, and its element at the top is the median.

For the priority queue interface, reimplement `priority_queue.py` adding two classes, namely, `MaxPriorityQueue` and `MinPriorityQueue`, that both derive from `PriorityQueue`, and add the right comparison function.

This implementation of `priority_queue.py` could contain

```
if __name__ == '__main__':
    max_pq = MaxPriorityQueue()
    min_pq = MinPriorityQueue()
    L = [13, 13, 4, 15, 9, 4, 5, 14, 4, 11, 15, 2, 17, 8, 14, 12, 9, 5, 6, 16]
    for e in L:
        max_pq.insert(e)
        min_pq.insert(e)
    print(max_pq._data[ : max_pq._length + 1])
    print(min_pq._data[ : min_pq._length + 1])
    for i in range(len(L)):
        print(max_pq.delete_top_priority(), end = ' ')
    print()
    for i in range(len(L)):
        print(min_pq.delete_top_priority(), end = ' ')
    print()
```

in which case testing this class would yield:

```
[None, 17, 16, 15, 13, 15, 5, 14, 13, 6, 14, 11, 2, 4, 4, 8, 12, 9, 4, 5, 9]
[None, 2, 4, 4, 5, 11, 4, 5, 9, 6, 13, 15, 13, 17, 8, 14, 15, 12, 14, 9, 16]
17 16 15 15 14 14 13 13 12 11 9 9 8 6 5 5 4 4 4 2
2 4 4 4 5 5 6 8 9 9 11 12 13 13 14 14 15 15 16 17
```

With this in place, the implementation of `median.py` could contain

```
if __name__ == '__main__':
    L = [2, 1, 7, 5, 4, 8, 0, 6, 3, 9]
    values = Median()
    for e in L:
        values.insert(e)
        print(values.median(), end = ' ')
    print()
```

in which case testing this class would yield:

```
2 1.5 2 3.5 4 4.5 4 4.5 4 4.5
```

2 A generalised priority queue

Reimplement `priority_queue.py` so as to insert pairs of the form `(datum, priority)`. If a pair is inserted with a datum that already occurs in the priority queue, then the priority is (possibly) changed to the (possibly) new value. This implementation of `priority_queue.py` could contain

```
if __name__ == '__main__':
    pq = PriorityQueue()
    L = [('A', 13), ('B', 13), ('C', 4), ('D', 15), ('E', 9), ('F', 4), ('G', 5), ('H', 14),
         ('A', 4), ('B', 11), ('C', 15), ('D', 2), ('E', 17),
         ('A', 8), ('B', 14), ('C',12), ('D', 9), ('E', 5),
         ('A', 6), ('B', 16)]
    for e in L:
        pq.insert(e)
    for i in range(8):
        print(pq.delete(), end = ' ')
    print()
    print(pq.is_empty())
```

in which case testing this class would yield:

```
B H C D A G E F
True
```