Python in the enterprise

Lab #1 report

Dawid Gerstel

March 9, 2016

1 Introduction

The aim of this lab exercise was to implement second order linear equations solver in Python using class-based approach. The application had to be based upon a class manager (`AppMgr`) that delegates procedures to other classes, namely: `InputReader`, `InputValidator` and `Solver`.

2 Project description

At first stage relationships among classes were taken into account. Since all classes were managed within the application manager `AppMgr` and relied on its data members – the three auxiliary classes were included in the `AppMgr` scope. Then, another class `OutputWriter` was added to the three to facilitate saving results to a text file.

The class `InputReader` is presented in the Listing 1. Its functionality comes down to the `__init__(self, fname)` method, which reads equation system parameters from a text file assuming its 2 first columns correspond to coefficients of the variables and the last column holds free terms. Subsequently the parameters are extracted to the `AppMgr.params` field.

Listing 1: `InputReader`

```
1  class AppMgr(object):
2    class InputReader(object):
3      def __init__(self, fname):
4        ''' reads 'fname' assuming formating:
5            a1 b1 c1
6            a2 b2 c2
7            ...
8            where the last column is a vector of free terms
9        '''
10       print "\nReading the input file: {}".format(fname)
11       with open(str(fname)) as f:
12         data = f.readlines()
13       AppMgr.params = [row.split() for row in data]
```

The `InputValidator` (Listing 2) avails of exception handling mechanism (function `is_number(s)`) that ensures the input values are numbers and throws `ValueError` otherwise, which leads to an adequate log message and the program stops.

If the input parameters are strings, therefore conversion to float is performed (line 11) provided that the parameter is indeed a number.

Listing 2: `InputValidator`

```
1  class InputValidator(object):
2    def __init__(self):
3      def is_number(s):
4        try:
5          float(s)
6          return True
7        except ValueError:
8          print "**[LOG]** At least one of input terms is not a
               number!!!\n"
9          exit()
10
11     AppMgr.params = [[float(elem) if is_number(elem) else
         None for elem in row] for row in AppMgr.params]
12     print "**[LOG]** The input parameters read:\n", AppMgr.
         params
```

The `Solver` from Listing 3 uses Cramer's method to solve the linear set of equations. It contains 3 methods calculating determinants (lines 2-22) as described by the method and checks if the system is both consistent and determinate (from line 29). Also, it has a method for printing feedback message plus solutions if any (line 24).

Listing 3: `Solver`

```
1  class Solver(object):
2    def calcDeter(self):
3      ''' calculates determinant of matrix A in A*x=b using
4          Cramer's method;
5          assumes 2nd order linear eq. system
6      '''
7      return AppMgr.params[0][0] * AppMgr.params[1][1] - \
8             AppMgr.params[0][1] * AppMgr.params[1][0]
9
10   def calcDeterX(self):
11     ''' calculates X determinant using Cramer's method
12         (assumes 2nd order linear eq. system)
13     '''
14     return AppMgr.params[0][2] * AppMgr.params[1][1] - \
15            AppMgr.params[0][1] * AppMgr.params[1][2]
16
17   def calcDeterY(self):
18     ''' calculates Y determinant using Cramer's method
19         (assumes 2nd order linear eq. system)
20     '''
21     return AppMgr.params[0][0] * AppMgr.params[1][2] - \
22            AppMgr.params[1][0] * AppMgr.params[0][2]
23
```

```
24      def printResult(self):
25        print AppMgr.solMsg
26        if AppMgr.sol[0] is not None:
27          print AppMgr.sol
28
29      def __init__(self):
30        D = self.calcDeter()
31        Dx = self.calcDeterX()
32        if not D: # either inconsistent or indeterminate
33          if not Dx:
34            AppMgr.sol = [None, None]
35            AppMgr.solMsg = "The system of eqs. is indeterminate"
36          else:
37            AppMgr.sol = [None, None]
38            AppMgr.solMsg = "The system of eqs. is inconsistent"
39        else:
40          AppMgr.sol = [Dx / D, self.calcDeterY() / D ]
41          AppMgr.solMsg = "The solution is:"
42        self.printResult()
```

The same feedback information as in the Solver's print method is saved to a text file, as described by the Listing 4.

Listing 4: `OutputWriter`

```
1   class OutputWriter(object):
2     def __init__(self, outfile):
3       print "**[LOG]** Writing to the output file: {}".format(
            outfile)
4       with open(outfile, 'w') as fout:
5         fout.writelines( AppMgr.solMsg + '\n' )
6         fout.writelines( str(AppMgr.sol) + '\n')
```

The following Listing 5 shows consecutive execution of the `AppMgr` initialisation that occurs by reading-in the input and output data files and then invoking the above described methods.

Listing 5: `AppMgr`

```
1  class AppMgr(object):
2  # ...
3    def __init__(self, DataFile, DataOutFile):
4      self.DataFile = DataFile
5      self.DataOutFile = DataOutFile
6      AppMgr.InputReader(self.DataFile)
7      AppMgr.InputValidator()
8      AppMgr.Solver()
9      AppMgr.OutputWriter(self.DataOutFile)
```

Listing 6 shows testing of the project by applying the application manager to 3 different cases: consistent and determinate system, inconsistent system, and indeterminate system. The program has successfuly recognised each situation and arrived at proper solutions in the first case.

Listing 6: `AppMgr`

```
1 #=== Main execution ===
2 appMgr = AppMgr("goodInput.txt", "goodOutput.txt")
3 appMgr = AppMgr("inconsistentInput.txt", "inconsistentOutput.
      txt")
4 appMgr = AppMgr("indeterminateInput.txt", "indeterminateOutput3
      .txt")
```