Assignment 3

COMP9021, Session 2, 2015

Aims: The purpose of the assignment is to:

- process command-line arguments in the style of Unix utilities;
- become familiar with trees;
- possibly implement recursive solutions;
- develop problem solving skills.

Command line arguments

Both programs will read command line arguments. To read those we import the `sys` module and process `sys.argv`, a list of strings whose first element is the name of the module being interpreted, whose second element is the first command line argument, whose third element is the second command line argument, etc. So if no command line argument is provided, then `sys.argv` is a list of length 1.

Submission

Your programs will be stored in files named `display_tree.py` and `roman_arabic.py`. Upload yours file using WebCMS. Assignments can be submitted more than once: the last version is marked. Your assignment is due by November 1 11:59pm.

Assessment

For each test, the automarking script will let your program run for 30 seconds.

For each of the two programs, up to one mark will reward good comments, good choice of names for identifiers and functions, readability of code, simplicity of statements, compactness of functions. This will be determined manually.

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.

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.

1 Displaying a tree

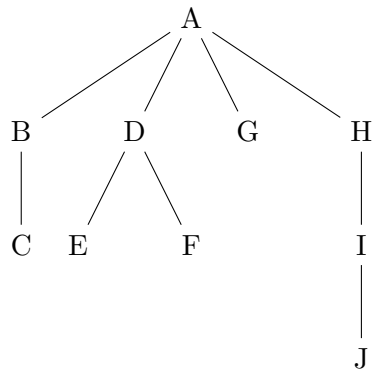
1.1 General description

The program will read from a `.txt` file a simple representation of a tree and write to a `.tex` file to display this tree graphically. The program will accept various command line arguments to choose between alternative representations.

The contents of `tree_1.txt` is the following (with spaces on nonblank lines clearly indicated).

```
A
␣␣␣␣B
␣␣␣␣␣␣␣␣C

␣␣␣␣D
␣␣␣␣␣␣␣␣E
␣␣␣␣␣␣␣␣F



␣␣␣␣G
␣␣␣␣H

␣␣␣␣␣␣␣␣I
␣␣␣␣␣␣␣␣␣␣␣␣J
```

If offers a simple, convenient representation of the tree depicted in `tree_1.pdf`, shown below:



The file tree_1.pdf has been obtained by running pdflatex on the file tree_1.tex, whose contents is:

```
\documentclass[10pt]{article}
\usepackage{tikz}
\usetikzlibrary{shapes}
\pagestyle{empty}

\begin{document}

\begin{center}
\begin{tikzpicture}
\node {A}
    child {node {B}
        child {node {C}}
    }
    child {node {D}
        child {node {E}}
        child {node {F}}
    }
    child {node {G}}
    child {node {H}
        child {node {I}
            child {node {J}}
        }
    };
\end{tikzpicture}
\end{center}

\end{document}
```

Your assignment will basically take as input a file such as `tree_1.txt`, and produce as output a file such as `file_1.tex`. It will have to check that the program is being run properly, that the contents of the input file is consistent with the representation of a tree, and produce an appropriate error message if needed.

1.2 Detailed description

A `.txt` file properly represents a tree if the following conditions are satisfied.

- The tree has at least 2 nodes (including the root), hence the file has at least 2 nonblank lines.
- The file can contain any number of blank lines, that is, lines containing nothing but spaces (possibly none).
- Each nonblank line in the file represents a unique node.
- The Control N character (`^N`) represents a node with no label; it can be used for any node except for the root.
- The Control E character (`^E`) represents an empty node; it can be used for any leaf.
- If a nonblank line does not consist of spaces and a unique `^E` or `^N` character, then the longest sequence of symbols that neither starts nor ends in a space represents the label of the node.
- The representation of each node can be only preceded by simple spaces (not tabs).
- The root can be preceded by any number x of spaces.
- A node on level n of the tree that is not the root is preceded by precisely $x + y * n$ simple spaces, where y is an arbitrary strictly positive constant.

For instance, in `tree_1.txt`, x is equal to 0 while y is equal to 4. An alternative representation of the same tree is given in `tree_1_a.txt`, for which x is equal to 3 and y is equal to 5 (and the blank lines have been removed):

```
␣␣␣A
␣␣␣␣␣␣␣␣B
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣C
␣␣␣␣␣␣␣␣D
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣E
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣F
␣␣␣␣␣␣␣␣G
␣␣␣␣␣␣␣␣H
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣I
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣J
```

Further examples are given as:

- the file `tree_2.txt`, that when the executable is given the option

  `-grow up`

  produces the file `tree_2.tex`—the `-grow` option should be followed by one of `down`, `up`, `left` or `right`;

- the file `tree_3.txt`, that when the executable is given the option

  `-nodestyle circle`

  produces the file `tree_3.tex`—the `-nodestyle` option should be followed by one of `rectangle`, `circle` or `ellipse`;

- the file `tree_4.txt`, that when the executable is given the options

  `-grow left -nodestyle ellipse`

  produces the file `tree_4.tex`;

- the file `tree_5.txt`, that when the executable is given the options

  `-nodestyle rectangle -grow down`

  produces the file `tree_5.tex`.

Note that providing the option `-grow down` does not yield the same pictorial representation of the tree as not providing the `-grow` option (trees grow down in both cases, but the left-right order of the branches is opposite). Also note that in the `.tex` files, the instruction to deal with the `-grow` option comes before the instruction to deal with the `-nodestyle` option, irrespective of the order which those options have been given in.

These examples should provide sufficient information on what your program should produce in case the input file contains valid data, for all valid uses of the possible options. But the program should check that options, if any, are valid, that a single filename ending in `.txt` is provided as last command line argument (after the options, if any), and that the file exists in the current directory. If that is the case then the program should check that the input file contain valid data, that indeed represent a tree, and otherwise output an error message that indicates the number of the **FIRST** nonblank line (amongst all nonblank lines) that is inconsistent with a representation of a tree. You do not have to check that the input file contains at least 2 nonblank lines (all test files will contain at least 2 nonblank lines), nor that the Control N character is never used to label the root, nor that the Control E character is only used to label a leaf. What has to be checked is only that the number of spaces before the first nonspace character on a given nonblank line is consistent with the representation of a tree in the form that has been described.

1.3 Sample outputs

Running

```
$ python3 display_tree.py tree_1.txt
$ python3 display_tree.py -grow up tree_2.txt
$ python3 display_tree.py -nodestyle circle tree_3.txt
$ python3 display_tree.py -grow left -nodestyle ellipse tree_4.txt
$ python3 display_tree.py -nodestyle rectangle -grow down tree_5.txt
```

should produce the files `tree_1.tex`, `tree_2.tex`, `tree_3.tex`, `tree_4.tex` and `tree_5.tex`, respectively. So the output file should have the same extension as the input file, except for the extension `.txt` being changed to `.tex`. To check your outputs on those examples, rename the provided files `tree_1.tex`, `tree_2.tex`, `tree_3.tex`, `tree_4.tex` and `tree_5.tex` and use `diff`, which should exit silently if your output is correct.

The following are examples of how your program should behave in case it is not run as expected, or the filename provided as argument does not exist in the current directory, or the data is incorrect. So if your program cannot process any data then it should output either `Incorrect invocation` when anything is wrong, except for the case where the filename provided as argument does not exist (but all the rest is ok), in which case the output is `No file named ... in current directory`.

```
$ python3 display_tree.py -nodestyle rectangle -lines dashed tree.txt
Incorrect invocation
$ python3 display_tree.py -nodestyle triangle tree_1.txt
Incorrect invocation
$ python3 display_tree.py -nodestyle tree_1.txt
Incorrect invocation
$ python3 display_tree.py -nodestyle rectangle tree.text
Incorrect invocation
$ python3 display_tree.py tree_1.txt -nodestyle rectangle
Incorrect invocation
$ python3 display_tree.py -nodestyle rectangle
Incorrect invocation
$ python3 display_tree.py -nodestyle rectangle tree_1.txt tree_2.txt
Incorrect invocation
$ python3 display_tree.py -nodestyle rectangle tree.txt
No file named tree.txt in current directory
$ python3 display_tree.py wrong_1.txt
Wrong number of leading spaces on nonblank line 7
$ python3 display_tree.py wrong_2.txt
Wrong number of leading spaces on nonblank line 4
$ python3 display_tree.py wrong_3.txt
Wrong number of leading spaces on nonblank line 5
$ python3 display_tree.py wrong_4.txt
Wrong number of leading spaces on nonblank line 6
```

2 Roman to arabic and arabic to roman

2.1 General presentation

You will design and implement a program that expects one, two or three command line arguments. When the program gets two command line arguments, the second one has to be `minimally`. When the program gets three command line arguments, the second one has to be `using`. The program outputs an error message about incorrect command line arguments if those conditions are not fulfilled.

In case exactly one command line argument is provided, the program outputs an error message about incorrect input if that argument is not a strictly positive number (whose representation should not start with 0) that can be converted to a roman number (being at most equal to 3999), or if that argument is not a valid roman number. Otherwise, the program performs the conversion and outputs the result.

In case exactly three command line arguments are provided, the program outputs an error message about incorrect symbols if the last command line argument is not a sequence of so-called generalised roman symbols, which is nothing but a sequence of pairwise distinct (uppercase or lowercase) letters, the classical roman symbols corresponding to the sequence `MDCLXVI`, whose rightmost element is meant to represent 1, the second rightmost element, 5, the third rightmost element, 10, etc. Otherwise, the program expects that the first command-line argument be a strictly positive number or a so-called generalised roman number, that is, a sequence of generalised roman symbols that can be decoded using the provided sequence of generalised roman symbols similarly to the way roman numbers are represented. The program outputs the error message about incorrect input if the first command line argument is not a proper number or if it is too large to be coded into a generalised roman number, or if the first argument cannot represent a generalised roman number. Otherwise, the program performs the conversion and outputs the result.

In case exactly two command line arguments are provided, the program tries and view the first command line argument as a generalised roman number with respect to some sequence of generalised roman symbols, and outputs the error message about incorrect input if that is not possible (so in particular, if the first command line argument contains occurrences of nonletters). Otherwise, the program tries and find the smallest arabic number that could be converted from such a generalised roman number. It outputs that arabic number followed by `using` and then the sequence of generalised roman symbols which makes that conversion possible; that sequence might contain occurrences of `_` for the generalised roman symbols that are not used in the generalised roman number.

2.2 Examples

2.2.1 Examples with incorrect command line arguments

```
$ python3 roman_arabic.py 123 by using ABC
I expect one, two or three command line arguments,
  the second one being "minimally" in case two of those are provided
  and "using" in case three of those are provided.
$ python3 roman_arabic.py 123 ussing ABC
I expect one, two or three command line arguments,
  the second one being "minimally" in case two of those are provided
  and "using" in case three of those are provided.
$ python3 roman_arabic.py ABCD minimally using ABCDE
I expect one, two or three command line arguments,
  the second one being "minimally" in case two of those are provided
  and "using" in case three of those are provided.
$ python3 roman_arabic.py ABCD minimaly
I expect one, two or three command line arguments,
  the second one being "minimally" in case two of those are provided
  and "using" in case three of those are provided.
```

2.2.2 Examples with an incorrect sequence of generalised roman numbers

```
$ python3 roman_arabic.py ADDB using ABCD01
The provided sequence of so-called generalised roman symbols is invalid,
  either because it does not consist of letters only
  or because some letters are repeated.
$ python3 roman_arabic.py ADDB using ABCDB
The provided sequence of so-called generalised roman symbols is invalid,
  either because it does not consist of letters only
  or because some letters are repeated.
```

2.2.3 Examples with exactly one command line argument

```
$ python3 roman_arabic.py 035
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py 4000
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py IIII
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py IXI
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py 1982
MCMLXXXII
$ python3 roman_arabic.py 3007
MMMVII
$ python3 roman_arabic.py MCMLXXXII
1982
$ python3 roman_arabic.py MMMVII
3007
```

2.2.4 Examples with exactly three command line arguments

```
$ python3 roman_arabic.py XXXVI using VI
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py XXXVI using IVX
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py XXXVI using XWVI
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py 900000000000 using AaBbCcDdEeFfGgHhIiJjKkLl
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py XXXVI using XVI
36
$ python3 roman_arabic.py XXXVI using ABCDEFGHJKMLXVI
36
$ python3 roman_arabic.py XXXVI using XABVI
306
$ python3 roman_arabic.py EeDEBBBaA using fFeEdDcCbBaA
49036
$ python3 roman_arabic.py 49036 using fFeEdDcCbBaA
EeDEBBBaA
$ python3 roman_arabic.py 899999999999 using AaBbCcDdEeFfGgHhIiJjKkLl
Aaaabacbdcedfegfhgihjikjlk
$ python3 roman_arabic.py ABCDEFGHIJKLMNOPQRST using AbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStT
1111111111111111111
$ python3 roman_arabic.py 1900604 using LAQMPVXYZIRSGN
AMAZING
```

2.2.5 Examples with exactly two command line arguments

```
$ python3 roman_arabic.py 0I minimally
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py ABAA minimally
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py ABCDEFA minimally
The input is not a valid arabic number,
  or is too large an arabic number,
  or is not a valid (possibly generalised) roman number,
  depending on what is expected.
$ python3 roman_arabic.py MDCCLXXXVII minimally
1787 using MDCLXVI
$ python3 roman_arabic.py MDCCLXXXIX minimally
1789 using MDCLX_I
$ python3 roman_arabic.py MMMVII minimally
37 using MVI
$ python3 roman_arabic.py VI minimally
4 using IV
$ python3 roman_arabic.py ABCADDEFGF minimally
49269 using BA_C_DEF_G
$ python3 roman_arabic.py ABCCDED minimally
1719 using ABC_D_E
```