Computing Science 1P

Laboratory Examination 2 2012-13

15-17 April 2013 **Time allowed:** 1 hour 50 minutes + advance preparation

Weight: the exam contributes 10% to the overall assessment for the CS1P course.

Threshold: students must attempt at least 75% of all assessments on the CS1P course in order to gain credit for the course. This exam contributes 10%.

Instructions

Before the exam

You should prepare a complete solution to the exam question given in this pack in advance of the exam. Put in as many hours as you need to get a properly working solution **that you fully understand**. You can of course type it into a machine and thoroughly test and debug it.

When you come into the exam, you will be able to bring nothing with you. We will provide you with another copy of the question along with some rough paper to work on. You need to be prepared to reconstruct your solution to the problem during the exam.

You may of course talk to others while preparing for the exam. Remember, however, that you will be on your own in the exam, and so you must understand your solution *thoroughly* before coming to the exam.

During the exam, the machines will be disconnected from the network and from your filestores. Note however that the Python Docs help in the Idle Help menu will be available. You should familiarise yourself with its contents, in case you get stuck during the exam with Python syntax.

There will be no access to the laboratory, other than to take the lab exam, at any time during the period 15-17 April inclusive. The lab may also be closed for short periods during the week before and on Thursday/Friday during the lab exam week to allow for preparation by the systems team.

Starting the exam

Complete the attendance slip and place it in front of you ***together with your matriculation/student ID card***.

Log in to the system using your special exam username and password as provided on the document in your examination pack. Use AMS to set up `LabExam2`. This will generate copies of the template program files in your exam account Workspace.

During the exam

As in a normal examination, communication between candidates during the laboratory exam is strictly forbidden.

A candidate who wishes to print a document during the exam should summon an invigilator, who will collect it from the printer and deliver it.

A candidate may leave early, but only after summoning an invigilator, who will ensure that submission has taken place. Any candidate who experiences a hardware or software problem during the examination should summon an invigilator at once.

At the end of the exam

Candidates should ensure that they have submitted their solution, using AMS, before the end of the examination.

Candidates must leave the laboratory quietly; the exam may still be in progress for other groups.

Lab Exam 2 Problem: Text Alignment

Modern word processors and typesetting systems allow users to specify the width of each line of a document as well as the alignment of text on the page.

The task of this exam is to write a program to read text from an input file, and be able to align it *left*, *right*, *centre*, and *fully justified* before writing it to an output file.

You can assume that the input text is aligned left, but not necessarily with the desired line width. Your program should allow a user to choose the maximum line width (in number characters) and the type of alignment. You can also assume that the input file contains a paragraph of text rather than, e.g., a whole book. In addition, you will be handling plain text and therefore a plain, fixed-width font for each character can be assumed.

The following sections provide further information about the main aspects of your program.

1. Loading and Saving Files

Your program should read text from a file supplied by the user, say `ffff.txt`. It should then ask the user for the desired line width (`n`) and alignment (left, right, centre, or justified) for the output. Aligned text should then be written to a file with same extension and a name of the form:
`ffff-aligned_xxxx-n.txt`
where `xxxx` is the type of alignment and `n` the width of each line.

For example, if `text.txt` is the input file, `n=40`, and chosen alignment is *right*, the output filename should be: `text-aligned_right-40.txt`

If the user chooses more than one alignments and/or line widths for the same input file, the corresponding number of output files with appropriate file names should be produced.

2. Aligning Text

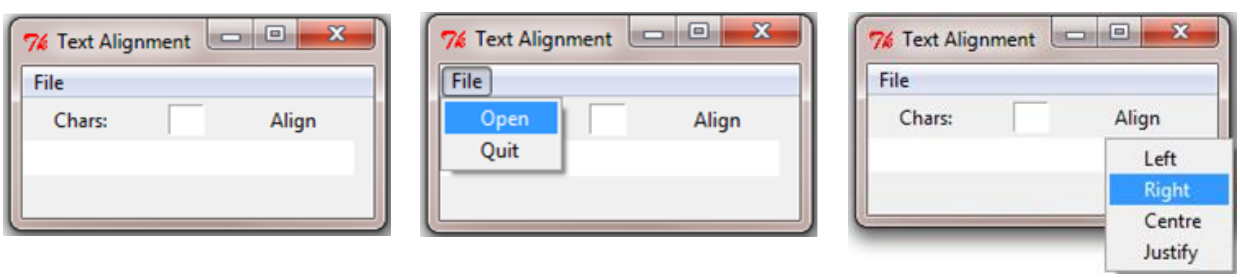
You should think carefully how to implement the different types of text alignment. Every single one of them should fit as many whole words as possible in the specified line width. You should not implement hyphenation. Instead, you should pad lines with white spaces to achieve the required text alignment.

You can assume a *fixed-width font*, and *whole-space* characters and white spaces. This is a simplification over what you get in modern word processing environments, such as, e.g., Microsoft Word.

For fully justified text, white spaces should be distributed as evenly as possible amongst the words of each line. The next page shows example input and output files for the different alignments and a given line width.

3. Graphical User Interface (GUI)

5/20 marks have been reserved for adding a GUI to your program. A skeleton for the GUI will be provided in the file `gui.py` and will be under the AMS directory. The GUI should include the following widgets, as illustrated in the screenshots and described below:



- A `File menu` with two `commands`, `Open` and `Quit`. `Open` launches the Windows file picker and lets you select an input file.
  You will find the `tkFileDialog.askopenfilename()` method useful for this.
  `Quit` destroys the top-level window.
- A `text entry`, where the user specifies the desired line width for the alignment, in number of characters. It is assumed that a reasonable width will be chosen, but you should enforce an upper bound on the lines' width from within your code.
- An `Align menu` with four `Radio Buttons` for the different types of alignment. When the user clicks on any of them, the relevant action should be taken, the appropriate output file should be created and then opened with a compatible Windows' program. You can assume text files, and you can safely use `Notepad` for this. You will find the following method useful:
  `subprocess.Popen('notepad %s' % filename)`
- A `Label` field to display relevant messages according to user actions, such as, e.g., incorrect filename, chosen line width, any other error messages you find relevant.

Most of the GUI layout is provided. You should complete `gui.py` to provide for the specified functionality and integrate it with your main text alignment program.

You can choose not to implement the GUI and instead provide a simple textual interface for running and testing your program. In this case you will be aiming for 15/20 marks.

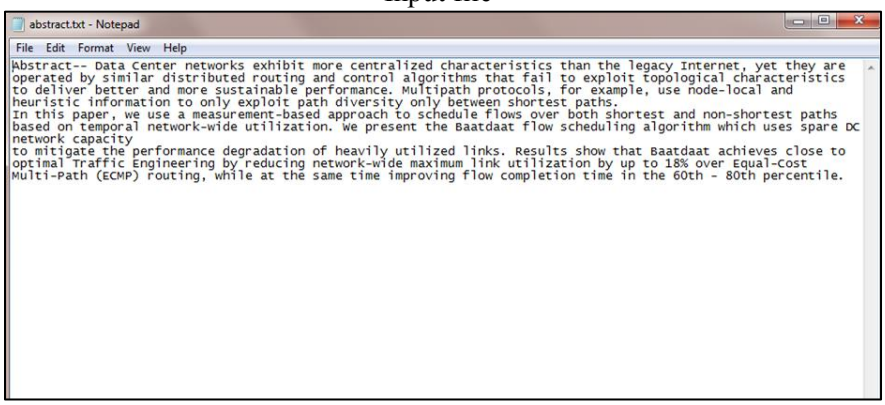
4. Testing

The AMS folder will include one input `.txt` file which you should use for testing. All *four* alignments should be performed on the input file with at least one specified `line width`. All output files produced should be submitted together with your solution.

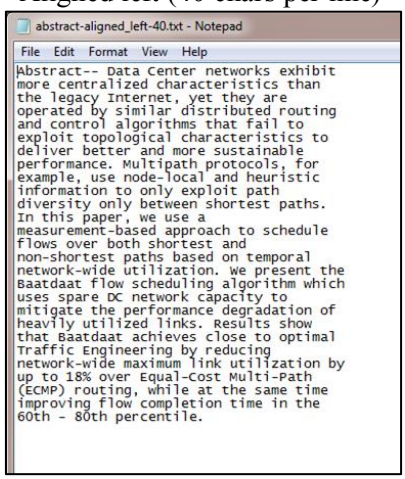
Extensive error checking (such as, e.g., for incorrect input, non-existing files, valid line width, etc.) should be part of GUI submissions only. If you chose to implement the textual interface, you can assume that input/output is correct, and you should focus on the correctness of the text aligning code.

5. Sample Input and Output Files

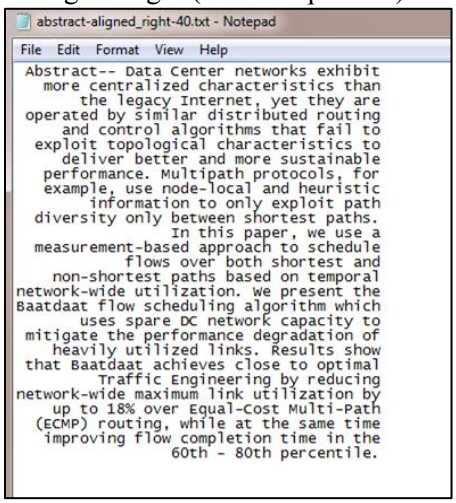
Input file



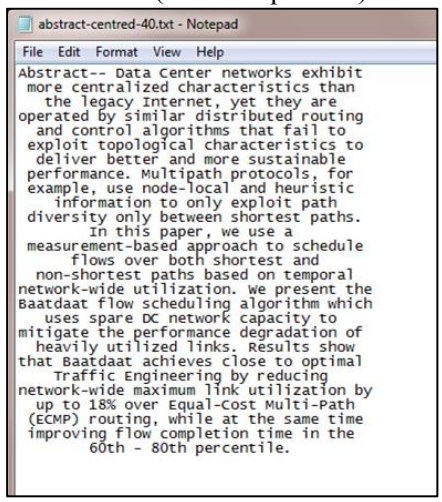
Aligned left (40 chars per line)



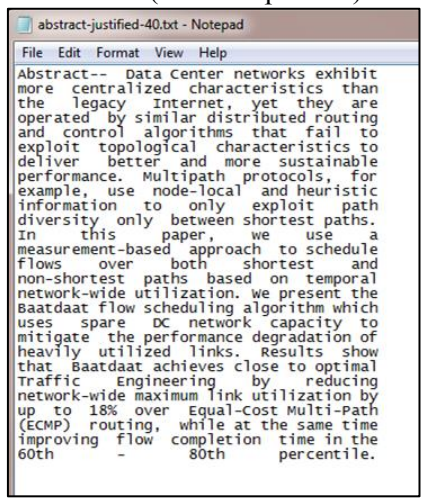
Aligned right (40 chars per line)



Centred (40 chars per line)



Justified (40 chars per line)



Specific Instructions

What you must do and what you must submit

You **may** use any standard Python modules, functions and methods. For example, the `split` and `join` functions from the `string` module (alternatively, the `split/join` methods of a string) are likely to be useful.

You **must** do your work in the file(s) **`text alignment.py`** (and **`gui.py`**, if you choose to implement the GUI too), provided as part of the AMS setup.

The following files which will be available under your AMS setup have also been made available on Moodle for you to start work on and check your solution:

- `text alignment.py`: placeholder for the main text alignment program
- `gui.py`: skeleton for the GUI. It contains most of MVC's *view* components as well as comments to aid you with the integration of the UI with the rest of the program
- `abstract.txt`: sample input file

Suggestions on how to progress

1. Choose appropriate data structure(s) to use and code/function(s) to process the input file
2. Think carefully about the design of your programme. Use the necessary functions for your code to be readable, your overall implementation efficient, and your error checking adequate.
3. If you choose to implement the GUI, carefully plan the interaction between `gui.py` and `text_alignment.py` in terms of function calls and parameter passing
4. If you choose to implement a textual interface, your code should all be in `text_alignment.py`. A textual menu should be implemented to enable the user to specify the required parameters and handle I/O

Marking guidelines

The exam will be marked out of 20. There are 5 marks reserved for the otherwise optional GUI.

- 10 marks for the implementation of the different text alignments: 2 marks for each of *left* and *right* alignments; 3 marks for each of *centre* and fully *justified* alignments.
- 5 marks for appropriate *file handling*, program *efficiency*, *clarity* and *structure*.

For GUI submissions:

- 2 marks for appropriate *controller* implementation, i.e., the design and interaction between widgets and functions.
- 3 marks for appropriate *model* functions, including extensive error checking and integration with functions in the main text_alignment.py.

The following aspects of the program will be taken into account in marking:

- use of appropriate and correct algorithms and Python control structures
- use of appropriate data structures
- correct Python syntax
- good programming style, including layout, use of explanatory comments, choice of identifiers, and appropriate use of functions
- correctness of the implementation on unseen test input