Assignment 3

COMP9021, Session 2, 2017

1 General presentation

This presentation refers to a number of text files that are provided together with this pdf.

Consider the two text files file_1_1.txt and file_1_2.txt:

```
$ cat -n file_1_1.txt
     1        A line to delete: 1
     2        A line to delete: 2
     3        A line that stays: 1
     4        A line that stays: 2
     5        A line to change: 1
     6        A line that stays: 3
     7        A line that stays: 4
     8        A line that stays: 5
     9        A line that stays: 6
    10        A line to delete: 3
    11        A line that stays: 7
    12        A line that stays: 8
    13        A line to change: 2
    14        A line to change: 3
    15        A line to change: 4
    16        A line to change: 5
    17        A line that stays: 9
$ cat -n file_1_2.txt
     1        A line that stays: 1
     2        A line to insert: 1
     3        A line that stays: 2
     4        A changed line: 1
     5        A line that stays: 3
     6        A line that stays: 4
     7        A line to insert: 2
     8        A line to insert: 3
     9        A line that stays: 5
    10        A line that stays: 6
    11        A line that stays: 7
    12        A line that stays: 8
    13        A changed line: 2
    14        A changed line: 3
    15        A changed line: 4
    16        A line that stays: 9
```

The Unix `diff` command can be applied to `file_1_1.txt` and `file_1_2.txt`:

```
$ diff file_1_1.txt file_1_2.txt
1,2d0
< A line to delete: 1
< A line to delete: 2
3a2
> A line to insert: 1
5c4
< A line to change: 1
---
> A changed line: 1
7a7,8
> A line to insert: 2
> A line to insert: 3
10d10
< A line to delete: 3
13,16c13,15
< A line to change: 2
< A line to change: 3
< A line to change: 4
< A line to change: 5
---
> A changed line: 2
> A changed line: 3
> A changed line: 4
```

Study this example to understand `diff`'s output.

- `d` is for *delete*, for commands of the form *`l_1`*`d`*`l_2`* with *`l_1`* and *`l_2`* two line numbers, or of the form *`l_1,l_2`*`d`*`l_3`* with *`l_1`*, *`l_2`* and *`l_3`* three line numbers.
- `a` is for *add*, for commands of the form *`l_1`*`a`*`l_2`* with *`l_1`* and *`l_2`* two line numbers, or of the form *`l_1`*`a`*`l_2,l_3`* with *`l_1`*, *`l_2`* and *`l_3`* three line numbers.
- `c` is for *change*, for commands of the form *`l_1`*`c`*`l_2`* with *`l_1`* and *`l_2`* two line numbers, or of the form *`l_1,l_2`*`c`*`l_3`* with *`l_1`*, *`l_2`* and *`l_3`* three line numbers, or of the form *`l_1`*`c`*`l_2,l_3`* with *`l_1`*, *`l_2`* and *`l_3`* three line numbers, or of the form *`l_1,l_2`*`c`*`l_3,l_4`* with *`l_1`*, *`l_2`*, *`l_3`* and *`l_4`* four line numbers.

`diff` identifies a *longest common subsequence* (LCS) of lines that are common to both files. It then computes the unique minimal set of commands that allows one to convert the first file into the second one (the `ed` editor can actually precisely do this). With the previous example, there is a unique LCS, consisting of the lines `A line that stays: 1`, `A line that stays: 2`, `A line that stays: 3` and `A line that stays: 4`.

Still the LCS is not always unique. Consider for instance the two text files `file_2_1.txt` and `file_2_2.txt`:

```
$ cat -n file_2_1.txt
     1        A line
$ cat -n file_2_2.txt
     1        A line
     2        A line
```

There are then 2 LCSs, as one can chose from `file_2_2.txt` either the first or the second line. Unix `diff` determines one LCS and yields the corresponding output:

```
$ diff file_2_1.txt file_2_2.txt
1a2
> A line
```

But another implementation could have output instead:

```
$ diff file_2_1.txt file_2_2.txt
0a1
> A line
```

For another example, consider the two text files `file_3_1.txt` and `file_3_2.txt`:

```
$ cat -n file_3_1.txt
     1        Line 1
     2        Line 2
     3        A line to go
     4        A line to go
     5        A line to go
     6        Line 3
     7        Line 4
     8        A line to go
$ cat -n file_3_2.txt
     1        A line to come
     2        A line to come
     3        Line 1
     4        Line 2
     5        A line to come
     6        A line to come
     7        Line 1
     8        Line 2
     9        A line to come
    10        A line to come
    11        A line to come
    12        Line 3
    13        Line 4
```

There are then 3 LCSs, as the unique occurrences of `Line 1` and `Line 2` in the first file can be matched with the first occurrences of `Line 1` and `Line 2` in the second file, or with the first occurrence of `Line 1` and the second occurrence of `Line 2` in the second file, or with the second occurrences of `Line 1` and `Line 2` in the second file.

Unix `diff` determines one LCS and yields the corresponding output:

```
$ diff file_3_1.txt file_3_2.txt
0a1,2
> A line to come
> A line to come
3,5c5,11
< A line to go
< A line to go
< A line to go
---
> A line to come
> A line to come
> Line 1
> Line 2
> A line to come
> A line to come
> A line to come
8d13
< A line to go
```

But another implementation could have output instead either

```
0a1,2
> A line to come
> A line to come
1a4,7
> Line 2
> A line to come
> A line to come
> Line 1
3,5c9,11
< A line to go
< A line to go
< A line to go
---
> A line to come
> A line to come
> A line to come
8d13
< A line to go
```

or

```
0a1,6
> A line to come
> A line to come
> Line 1
> Line 2
> A line to come
> A line to come
3,5c9,11
< A line to go
< A line to go
< A line to go
---
> A line to come
> A line to come
> A line to come
8d13
< A line to go
```

One could want to output the LCS, common to both files, with `...` for longest nonempty sequences of lines that are not part of the LCS, w.r.t. the first file, or w.r.t. the second file.

For the Unix `diff` applied to `file_1_1.txt` and `file_1_2.txt`, that would be, w.r.t. `file_1_1.txt`:

```
...
A line that stays: 1
A line that stays: 2
...
A line that stays: 3
A line that stays: 4
A line that stays: 5
A line that stays: 6
...
A line that stays: 7
A line that stays: 8
...
A line that stays: 9
```

and w.r.t. file_1_2.txt:

```
A line that stays: 1
...
A line that stays: 2
...
A line that stays: 3
A line that stays: 4
...
A line that stays: 5
A line that stays: 6
A line that stays: 7
A line that stays: 8
...
A line that stays: 9
```

For the Unix diff applied to file_2_1.txt and file_2_2.txt, that would be, w.r.t. file_2_1.txt:

```
A line
```

and w.r.t. file_2_2.txt:

```
A line
...
```

For the Unix diff applied to file_3_1.txt and file_3_2.txt, that would be, w.r.t. file_3_1.txt:

```
Line 1
Line 2
...
Line 3
Line 4
...
```

and w.r.t. file_3_2.txt:

```
...
Line 1
Line 2
...
Line 3
Line 4
```

2 Task specifications

Write a program stored in a file named `diff.py` that implements three classes,

- `DiffCommands`,
- `DiffCommandsError` and
- `OriginalNewFiles`,

the second one deriving from `Exception`.

`DiffCommands` does not provide any method in its public interface. It builds a `DiffCommands` object from a text file meant to store a plausible sequence of diff commands, one command per line, without any space on any line, and without any extra line. In case the text file does not satisfy those conditions, the `__init__()` method of `DiffCommands` raises a `DiffCommandsError` error with as message `Cannot possibly be the commands for the diff of two files`. Seven files of this type are provided (`wrong_1.txt` to `wrong_7.txt`). Three files from which you can build a `DiffCommands` object are provided:

- `diff_1.txt`, which stores the only possible sequence of diff commands for `file_1_1.txt` and `file_1_2.txt`;
- `diff_2.txt`, which stores one of the two possible sequences of diff commands for `file_2_1.txt` and `file_2_2.txt`;
- `diff_3.txt`, which stores one of the three possible sequences of diff commands for `file_3_1.txt` and `file_3_2.txt`.

`DiffCommands` implements the `__str__()` function so that `print()` can be used to output the diff commands of the object under consideration.

`OriginalNewFiles` provides a user interface with 4 methods:

- `output_diff()`
- `output_unmodified_from_original()`
- `output_unmodified_from_new()`
- `get_all_diff_commands()`

The following example of interaction shows how the methods are supposed to be called and what they are supposed to return. The only point which might not be obvious from that example is that in the sequence of diff commands output by the method

`OriginalNewFiles.pair_of_files.get_all_diff_commands()`

the diff commands are lexicographically ordered.

```
$ python3
...
>>> from diff import *
>>> DiffCommands('wrong_1.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('wrong_2.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('wrong_3.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('wrong_4.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('wrong_5.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('wrong_6.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('wrong_7.txt')
...
diff.DiffCommandsError: Cannot possibly be the commands for the diff of two files
>>> DiffCommands('diff_1.txt')
<diff.DiffCommands object at ...>
>>> diff_1 = DiffCommands('diff_1.txt')
>>> print(diff_1)
1,2d0
3a2
5c4
7a7,8
10d10
13,16c13,15
>>> diff_2 = DiffCommands('diff_2.txt')
>>> print(diff_2)
1a2
>>> diff_3 = DiffCommands('diff_3.txt')
>>> print(diff_3)
0a1,2
3,5c5,11
8d13
>>> pair_of_files = OriginalNewFiles('file_1_1.txt', 'file_1_2.txt')
```

```
>>> pair_of_files.is_a_possible_diff(diff_1)
True
>>> pair_of_files.is_a_possible_diff(diff_2)
False
>>> pair_of_files.is_a_possible_diff(diff_3)
False
>>> pair_of_files.output_diff(diff_1)
1,2d0
< A line to delete: 1
< A line to delete: 2
3a2
> A line to insert: 1
5c4
< A line to change: 1
---
> A changed line: 1
7a7,8
> A line to insert: 2
> A line to insert: 3
10d10
< A line to delete: 3
13,16c13,15
< A line to change: 2
< A line to change: 3
< A line to change: 4
< A line to change: 5
---
> A changed line: 2
> A changed line: 3
> A changed line: 4
>>> pair_of_files.output_unmodified_from_original(diff_1)
...
A line that stays: 1
A line that stays: 2
...
A line that stays: 3
A line that stays: 4
A line that stays: 5
A line that stays: 6
...
A line that stays: 7
A line that stays: 8
...
A line that stays: 9
```

```
>>> pair_of_files.output_unmodified_from_new(diff_1)
A line that stays: 1
...
A line that stays: 2
...
A line that stays: 3
A line that stays: 4
...
A line that stays: 5
A line that stays: 6
A line that stays: 7
A line that stays: 8
...
A line that stays: 9
>>> pair_of_files.get_all_diff_commands()
[<diff.DiffCommands object at ...>]
>>> diffs = pair_of_files.get_all_diff_commands()
>>> len(diffs)
1
>>> print(diffs[0])
1,2d0
3a2
5c4
7a7,8
10d10
13,16c13,15
>>> pair_of_files = OriginalNewFiles('file_1_2.txt', 'file_1_1.txt')
>>> diffs = pair_of_files.get_all_diff_commands()
>>> len(diffs)
1
>>> print(diffs[0])
0a1,2
2d3
4c5
7,8d7
10a10
13,15c13,16
>>> pair_of_files = OriginalNewFiles('file_1_1.txt', 'file_1_1.txt')
>>> diffs = pair_of_files.get_all_diff_commands()
>>> len(diffs)
1
>>> print(diffs[0])

>>> pair_of_files = OriginalNewFiles('file_2_1.txt', 'file_2_2.txt')
```

```
>>> pair_of_files.is_a_possible_diff(diff_1)
False
>>> pair_of_files.is_a_possible_diff(diff_2)
True
>>> pair_of_files.is_a_possible_diff(diff_3)
False
>>> pair_of_files.output_diff(diff_2)
1a2
> A line
>>> pair_of_files.output_unmodified_from_original(diff_2)
A line
>>> pair_of_files.output_unmodified_from_new(diff_2)
A line
...
>>> diffs = pair_of_files.get_all_diff_commands()
>>> len(diffs)
2
>>> print(diffs[0])
0a1
>>> print(diffs[1])
1a2
>>> pair_of_files = OriginalNewFiles('file_2_2.txt', 'file_2_1.txt')
>>> diffs = pair_of_files.get_all_diff_commands()
>>> len(diffs)
2
>>> print(diffs[0])
1d0
>>> print(diffs[1])
2d1
>>> pair_of_files = OriginalNewFiles('file_3_1.txt', 'file_3_2.txt')
>>> pair_of_files.is_a_possible_diff(diff_1)
False
>>> pair_of_files.is_a_possible_diff(diff_2)
False
>>> pair_of_files.is_a_possible_diff(diff_3)
True
```

```
>>> pair_of_files.output_diff(diff_3)
0a1,2
> A line to come
> A line to come
3,5c5,11
< A line to go
< A line to go
< A line to go
---
> A line to come
> A line to come
> Line 1
> Line 2
> A line to come
> A line to come
> A line to come
8d13
< A line to go
>>> pair_of_files.output_unmodified_from_original(diff_3)
Line 1
Line 2
...
Line 3
Line 4
...
>>> pair_of_files.output_unmodified_from_new(diff_3)
...
Line 1
Line 2
...
Line 3
Line 4
>>> diffs = pair_of_files.get_all_diff_commands()
>>>  len(diffs)
3
>>>  print(diffs[0])
0a1,2
1a4,7
3,5c9,11
8d13
>>>  print(diffs[1])
0a1,2
3,5c5,11
8d13
>>>  print(diffs[2])
0a1,6
3,5c9,11
8d13
```

```
>>> pair_of_files = OriginalNewFiles('file_3_2.txt', 'file_3_1.txt')
>>> diffs = pair_of_files.get_all_diff_commands()
>>> len(diffs)
3
>>> print(diffs[0])
1,2d0
4,7d1
9,11c3,5
13a8
>>> print(diffs[1])
1,2d0
5,11c3,5
13a8
>>> print(diffs[2])
1,6d0
9,11c3,5
13a8
```

2.1 Submission

Your programs will be stored in a file named `diff.py`. After you have developed and tested your program, upload your files using Ed. Assignments can be submitted more than once: the last version is marked. Your assignment is due by June 4, 11:59pm.

2.2 Assessment

The assignment is worth 10 marks. the automarking script will allocate 30 seconds to each run of your program.

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.

2.3 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.