

## Mandatory Assignment 5

Regular expressions (30 + 15 points/40 + 5 points)

University of Oslo - INF3331/INF43331

Fall 2018

All solutions should be stored in a directory called `assignment5` in your private repository. In this assignment, INF3331 students may solve the assignments for INF4331 to earn extra points.

It is also important to note that the assignment expects you to use regular expressions. You can not expect a full score if you use something else to solve the regex parts of the assignment.

Your code should also be well documented. All functions should have docstrings. Include a readme file with information and examples on how to run your scripts.

### 5.0 Background info on syntax highlighting (0 points)

When you use a good editor to edit a file with Python code, it will color it to make it easier to read for humans. Generally speaking comments will have a color to make them stand out from the rest of the text, so it is easier for us to separate them from the “active” parts of the program.

```
from my_unit_testing import unittest

def better_addition(a, b, num_checks=2):
    """Returns sum of a, b, but double checks answer several times."""
    sum_computations = [a + b for n in range(num_checks)]

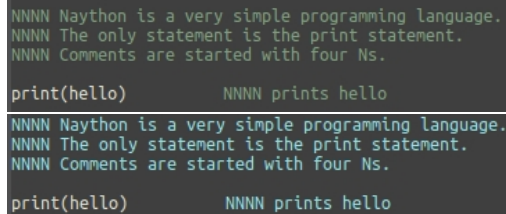
    for n in range(num_checks):
        if sum_computations[n] != sum_computations[n-1]:
            print("Hang on, let me recheck that")
            return better_addition(a, b, num_checks)

    return sum_computations[0] # if all computations match, return whichever
```

In the first part of this assignment, you will create a program which does that, and make it extensible to use different color schemes and support different languages. It should read a regex dictionary from a file which in a sense specifies what parts of the language should be colored, and a color theme from another file which specifies what colors should be used.

For examples, take a look at the example files `naython.syntax`, `naython1.theme`, and `naython2.theme` which specifies the syntax of a very simple “programming” language and gives two different color themes to color it by. When your

program is used to color `hello.ny`, your output should look like one of the two below, depending on which theme file you used:



```
NNNN Naython is a very simple programming language.
NNNN The only statement is the print statement.
NNNN Comments are started with four Ns.

print(hello)           NNNN prints hello

NNNN Naython is a very simple programming language.
NNNN The only statement is the print statement.
NNNN Comments are started with four Ns.

print(hello)           NNNN prints hello
```

**Note:** Regular expressions can be tricky to write, and they can get hairy very quickly. Your code is not expected to handle all kinds of edge cases, but try to get it to work in the majority of cases.

In addition to being difficult to write, regular expressions are even harder to read. Documentation for your regular expressions should also be given, in the cases where it's not obvious how they work.

## 5.1 Syntax highlighting (7 points)

Create a program which takes as input a regex , a color theme and a file and outputs the file with appropriate colors to standard out. Your program should be callable from the command line as `python3 highlighter.py syntaxfile themefile sourcefile.to_color` in the manner described above.

- Your `.syntax` files should have lines of the form `regex: name` where `regex` is a quoted string specifying a regex, and `name` is some arbitrary alphabetical string giving some name to the thing specified by the regex. See `naython.syntax` for an example.
- Your `.theme` files should have lines of the form `name: color_sequence` where `name` is one of the names specified in the matching `.syntax` file, and `color_sequence` is some bash color sequence. (i.e. something which would be valid if you did `"\033[{}m".format(color_sequence)`)
- Information on printing in color can be found in [http://misc.flogisoft.com/bash/tip\\_colors\\_and\\_formatting](http://misc.flogisoft.com/bash/tip_colors_and_formatting), and an example in `coloring_example.py` in the student resources repository.

Note that you need to be careful with printing in color when two regexes “overlap”. For example, if you for example have decided to color Python strings blue and newline characters green, coloring `‘Line 1 \n Line 2’` properly means printing first the color sequence for blue, then `‘Line 1` , then the color sequence for green, then `\n`, then the color sequence for blue, then `Line 2’`. So your program needs to “notice” that it needs to re-insert the blue color sequence in the middle of the string.

Name of file: `highlighter.py`.

## 5.2 Python syntax (INF3331: 5 points + 5 bonus points) (INF4331: 10 points)

Create a regex dictionary and color theme for Python. Your files should be usable by your program and follow the format specified in 5.0 - look at the naython examples for guidance. Choose at least 7 (INF4331: at least 10. INF3331: For bonus points, choose at least 10.) pieces of Python syntax from the below list and include them in your dictionary. Then create a second color theme. The color themes should be different, but you are free to use whichever colors you think look nicest.

Make sure to document which items you have chosen, in case it is not 100 % clear from the theme file.

- Comments
- Function definitions
- Class definitions
- Strings
- Imports
- “Special” statements `None`, `True`, `False`
- Variable assignments
- Decorators
- Try/except
- for-loops
- while-loops
- if/elif/else blocks
- Something else you feel is missing (be reasonable)

Name of files: `python.syntax`, `python.theme`, `python2.theme`

## 5.3 Syntax for your favorite language (INF3331 and INF4331, up to 5 bonus points)

For 5 bonus points, repeat 5.2 for a different language.

Use your best judgment on how much work you should expect to do here. If you make a complete syntax highlighter for whitespace (the language), that is very cute, but you should not expect to get 5 points. On the other hand, if you manage to color a decent (but not all) subset of Java, that is probably enough. The language should be sufficiently different from Python.

Name of files: `favorite_language.syntax`, `favorite_language.theme`

#### 5.4 `grep` (INF3331: 5 points + 5 bonus points. INF4331: 10 points)

Create a `grep`-like utility. It should take a filename, along with a filename, and print all lines where there the regex matches on one part of the line. The regex does not have to match from the start of the line, if you a line `aabbbccccc` and the regex is `bb`, the line should be printed. Further, it should be possible to color the *matching* parts of the lines by using a `--highlight` flag.

For INF4331/bonus, it should be possible to take an “arbitrary” number of regular expressions, and the script should be able to get all of these. The matching parts should be colored according to what regex they match. However, it is enough to choose a few colors, and cycle between them, to allow several regular expressions.

You are encouraged to use code previously written in this assignment (i.e. the highlighter). Therefore it is recommended that you try to make your code reusable in 5.1. You should also try to avoid duplicate code. For parsing of the command line arguments, the `argparse` module is recommended.

It is not expected that your `grep.py` script should work exactly as the UNIX utility. For instance, your implementation is expected to use Python’s regex syntax.

Name of file(s): `grep.py` + additional files if necessary.

#### 5.5 `superdiff` (10 points)

The standard utility `diff` takes two files as input, and outputs a file containing all changes which have to be made to the first file to make it into the second file. In this problem, you will create your own implementation of the standard `diff` utility. For convenience, your implementation should take two filenames from the command line, and treat the first as the “original” version of a file, and the second as a modified version.

It should then go through the files line by line, and if a line has not been modified, print it with a `0` in front, if it has been added, print it with a `+` in front, and if it has been deleted, print it with a `-` in front. If a line has been modified, treat it as being a deletion of the original line, and then an addition of the modified line.

Note that “line by line” is a bit misleading, as it is not enough to simply compare the two files line by line. Instead, you should try to find what the files have in common, to try to find the minimal modifications necessary to get the “modified file”. There is however some ambiguity inherent to the problem. For example, if the original has the lines “A”, “B” and the modified version has the lines “B”, “A”, was “A” deleted and later inserted, or was “B” inserted and later deleted? There are a lot of strange edge cases, so don’t be too worried about making an implementation which handles absolutely all of them.

You should try to make sure that your program is “reasonable” (i.e. if the original is “A”, “B”, “C”, “D”, “E”, “F” and the modified file is “B”, “C”, “D”, “E”, “F”, “G”, your program should ideally output that “A” has been deleted and “G” has been inserted at the end, and not say “everything in the first file was deleted, then everything in the second file was inserted”).

Name of file: `diff.py`

## 5.6 Coloring diff (3 points)

In this assignment, you will take the output from `diff.py` and color it so that additions become green, deletions become red and no-change lines aren’t colored in any special way. As you might notice, this is similar to what you did before, so instead of making a new script, make your program as a syntax file and color theme for your syntax highlighter.

Your syntax and theme should work so that if the output from `diff.py` is stored in the file `diff_output.txt`, the call `python3 highlighter.py diff.syntax diff.theme diff_output.txt` should produce the desired colored diff.

Name of files: `diff.syntax`, `diff.theme`