COM6115: Lab Class 5A

Regular Expressions

This lab provides an opportunity to practise writing and using regular expressions in Python.

PLEASE NOTE: to complete these exercises, you will need to refer to the "Extended Presentation" slides on Regular Expressions in the lab folder.

Regular expressions are a powerful and useful tool in programming, but they are not easy to get to grips with. It will be useful to spend some time practicing using them, as they are likely to be helpful for other things you work on in the future.

You may find it helpful to consult a "regex checker" such as the one at Regex101.com while working with regular expressions.

Exercises

Understanding the starter code

1. Download the files `regexes_STARTERCODE.py` and `RGX_DATA.html` from the lab class folder on Blackboard. Save the latter as a **single file with HTML tags included**. (*Beware*: if you open the HTML file in certain browsers and save it from there, then the browser may play tricks on you and the file may not work for the exercises.)

2. Run the STARTERCODE script. You will see that it reads the HTML data file, and prints each line to the screen, preceded by a separator line consisting of a long row of dashes (at the end of which the line number is printed). Thus, this begins:

```
---------------------------------------------------------------------------- [1]
TEXT: <html>
---------------------------------------------------------------------------- [2]
TEXT: <head>
---------------------------------------------------------------------------- [3]
TEXT: <title>COM3290: Symbolic Reasoning</title>
---------------------------------------------------------------------------- [4]
TEXT: </head>
```

3. The STARTERCODE script contains a line that compiles a regular expression (RE) as follows:

```
testRE = re.compile('(logic|sicstus)', re.I)
```

   This RE matches strings that are `"logic"` or `"sicstus"`, and does so in a **case-insensitive** manner (due to the `re.I` modifier — see slide 23 of the Extended Presentation).

   The STARTERCODE script matches this RE against each line of the file, and if a match is found, prints out the matching substring, which is the "group 1" of the match object, using the following code:

```
m = testRE.search(line)
if m:
    print('** TEST-RE:', m.group(1))
```

We can see that there is a successful match for line 40 of the data file, as the following is printed to screen:

```
   ---------------------------------------------------------------------------- [40]
   TEXT: <li> [<A HREF=lectures/lecture_logic_intro1.pdf>1up</A>] &nbsp SICStus Pr...
** TEST-RE: logic
   ---------------------------------------------------------------------------- [41]
```

If you study the input line of text, however, you'll see there are **several** substrings that would match the RE, and this method only catches the first.

4. Observe that the STARTERCODE script contains a further piece of code (which is *commented out*) as follows:

```
mm = testRE.finditer(line)
for m in mm:
    print('** TEST-RE:', m.group(1))
```

The `finditer` method is an **all matches** method. You may already have used the `findall` method, which returns a list of the multiple matching *strings*. `finditer` instead returns an **iterator**, one that sequentially yields a **match object** for each of the multiple matches. (See slides 19 and 20 of the Extended Presentation.) In the above code, these match objects are accessed (via a `for`-loop) so that the group-1s can be printed.

If you **uncomment** the above code (and also **comment out** the simpler alternative code fragment used earlier), you see that the screen output for input line 40 now prints out multiple matches, i.e. as in:

```
   ---------------------------------------------------------------------------- [40]
   TEXT: <li> [<A HREF=lectures/lecture_logic_intro1.pdf>1up</A>] &nbsp SICStus Pr...
** TEST-RE: logic
** TEST-RE: SICStus
** TEST-RE: Logic
   ---------------------------------------------------------------------------- [41]
```

Extending the starter code to match HTML elements

Your task is to write Python regular expressions to recognise certain patterns in the HTML text file.

Add code to the STARTER script to compile REs for these patterns (assigning them to variables with **meaningful names**), and to apply these REs to each line of the file, printing out the matches that are found — in the manner illustrated above.

1. Write a pattern for recognising HTML tags, which are then printed out as `"TAG: tag-string"` (e.g. `"TAG: b"` for tag <b>).

   For simplicity, assume that the open and close angle brackets (`<`,`>`) of each tag will always appear in the same line of text.

   A first attempt might use the regex `"<.*>"` where "`.`" is the predefined character class symbol matching **any** character. Try this out, to find out why this is **not** a good solution. Write a better solution that fixes this problem. (Speak to a demonstrator if you don't see the problem, or see how to fix it.)

2. Modify your code so that it distinguishes opening and closing tags (e.g. `<p>` vs. `</p>`), printing them as `OPENTAG` and `CLOSETAG`.

3. Note that some HTML tags have *parameters*, e.g. the 'extra bits' after the tag name in:

   ```
   <table border=1 cellspacing=0 cellpadding=8>
   ```

   Make sure your pattern for open-tags works for tags both with and without parameters, i.e. successfully finds and prints the tag label.

   Now extend your code so that it prints both the open tag label and the parameters, e.g. in the manner

   ```
   OPENTAG: table
      PARAM: border=1
      PARAM: cellspacing=0
      PARAM: cellpadding=8
   ```

4. In regexes, **backreferences** can be used to indicate that the substring that matched an earlier part of a regex should appear again. (See slide 25 of the Extended Presentation.)

   A backreference has the form `\N` (where `N` is a positive integer), and refers back to the text matched by the *Nth* group of the regex. (Note that a *raw string* (`r".."`) must be used when backreferences are present — see slide 15 of the Extended Presentation.)

   For example, a regex such as:

   ```
   r" (\w+) \1 "
   ```

   would match only if exactly the character string that matched the group `(\w+)` appeared again the position where the backref `\1` appeared. This could match the string "`kick the the ball`", for example, where "`the`" appears twice.

   Write a pattern using backreferences that will match when a line contains **paired** open and close tags, e.g. as in `<b>bold stuff</b>` and prints the material that was between, e.g. as `PAIR [b]: bold stuff`.

5. Observe in the file how *links* are specified in HTML, i.e. using a tag such as:

   `<a href=http://www........html>`

   where the target URL is given as the value of the `href` parameter, and with the "link" text seen in the viewed page appearing between such a tag and a corresponding close tab `</a>`. Write a case for recognising and printing URL expressions.

6. Consider that we might want to create a script that performs HTML stripping, i.e. which takes a file of HTML, and returns a file of *plain text* from which all HTML tags have been stripped out.

   We shall not attempt this here, but instead consider the simpler case of just *deleting* any HTML tags that we find in any line from our input data file.

   Consult slide 24 of the Extended Presentation, on using the RE `.sub` method to perform **string substitutions**. Extend your code so that (after any other matches have been done), any HTML tags are deleted, by using string substitution to replace them with the empty string `''`. You should be able to the RE you have already defined for **recognising** HTML tags as the basis for doing this.

   Print the resulting text to screen as `STRIPPED: .......`

Further ideas

1. Now that you have learned more about regular expressions, you could revisit the earlier labs in which we used them. Can you think of any ways to improve the way you extracted POS tags or tokenised text into words?

2. If you want to practice recognising what regular expressions are doing, the puzzles on regexcrossword.com could be worth a look.

3. This video from Stand-up Maths shows an unusual application of regular expressions to identify whether a number is prime, and includes explanations of several RegEx concepts.