Lab 10

Regular expressions

[Compulsory]

Author: Ragnar Nohre 2017, modified and moved to English Johannes Schmidt 2018-2022

So called *regular expressions* (regex) are used to search in texts for strings or patterns that fulfill certain criteria. This lab's purpose is to gain experience in using regex in order to extract information from websites.

10.1 Introduction

A *regex* (or *regexp*) is a small (but complicated) text string that describes a matching condition for string matching. With such regex on can for instance describe the formal rules of how email addresses look like and then search for email addresses in a certain text document.
Regex exist in principle since 1956. In Unix-based operating systems you find a command (egrep) that can search in files for text strings that match a certain regex. Regex are also included in many programming languages nowadays, e.g. python, javascript. Since 2011 it is also part of C++.

10.1.1 Readings and links

This lab should be complemented with reading of lecture slides, *docs.python.org/3/library/re.html* and a look at

- regex101.com
- regexr.com

- regexpal.com

10.2 Introductory experiments

It is recommended to learn regex with python in interactive mode. Type in the following.

```
import re
txt = "Hanoror bought 5 portions of orange soil for 13.50 EUR."
re.findall("or",  txt)
```

A short description of what these lines do:

1. The first row imports the regex-module `re`. If you enter `dir(re)` you will see what it contains.
2. The next line initialises `txt` with some test data to train on.
3. Then we call the `findall`-function from the `re` module. Write `help(re.findall)` and read the small description. We search for the text *or* in the longer text `txt`. The function returns a list with all occurrences. The list should contain 5 `or`-strings (because *Hanoror*, *portions*, *orange* and *for* together contain 5 times that string).

In the above example the regex was the simple string 'or'. In the following we experiment with other expressions. It can be very useful to use the arrow-up/down keys to recall previously entered statements in order to reuse and modify them.

10.2.1 Dot means joker

A dot in a regex is a kind of 'joker' that matches *all* characters (except a line break '`\n`'). Try to search for a single dot. All characters should match:

```
re.findall(".", txt)
```

Try also to add a dot after *or*:

```
re.findall( "or.", txt)
```

This regex searches for strings that start with *or* and continue with any character. Observe that you get four matches. Maybe you expected five, because 'Norori' contains both 'oro' and 'ori'. But `findall` finds only non-overlapping strings.

How do we search for a real, ordinary dot? Search for `\.` instead. Let us for instance search for strings that consist of two arbitrary characters followed by a dot:

```
    re.findall("..\.", txt)
```

You should get two matches.

10.2.2 r-prefix

Sometimes it is problematic to write the \-character in a regex because python's preprocessor can interpret it as a non-printable character. In order to avoid this one should always prefix a regex with *r* (r stands for *raw*). Write

```
  print("Hello\nWorld")
```

and compare with

```
  print(r"Hello\nWorld")
```

and you will understand the issue.

10.2.3 Other special characters

We have seen the dot has a special meaning. The are a number of other characters with a special meaning.

word-characters

Run this:

```
    re.findall(r"\w", txt)
```

`\w` matches clearly all characters, except typical separator characters (space, line break, line feed, tabulator). The rule to remember is *w as in word*. But also digits and other characters allowed in variables names count. Replace the `\w` by a capital `\W` and see what happens:

```
    re.findall(r"\W", txt)
```

This is obviously the exact opposite of `\w`.

Generally, if you capitalize a special character, it has the opposite meaning (`\W` means: **not** a word character).

digit-characters

Try `\d` and `\D`

```
re.findall(r"\d", txt)
re.findall(r"\D", txt)
```

The rule to remember is *d as in digit.*

space-character (separator-characters)

Try `\s` and `\S`

```
re.findall(r"\s", txt)
re.findall(r"\S", txt)
```

The rule to remember is *s as in space* or *s as in separator*, because also line break, line feed, and tabulator (`'\n'`, `'\r'`, `'\t'`) count.

10.2.4 User defined sets

With square brackets one can easily create a *set* of searchable characters. For instance, the following finds all consonants (English language):

```
re.findall(r"[bcdfghjklmnpqrstvxz]", txt)
```

So, `[bcdfghjklmnpqrstvxz]` is just a shortcut for `(b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|x|z)`.

If the circumflex character (^) is placed as the very first character, it means the negation of the indicated characters. The following regex matches all characters that are not a vowel (English language):

```
re.findall(r"[^aeiouy]", txt)
```

If you want the circumflex character to be part of the set, simply do not place it first.
If one places the minus character (-) between two characters in a user defined set, it means *to*. The following regex finds all occurrences of digits and the letters a,b,c, . . . f:

```
re.findall(r"[0-9a-f]", txt)
```

If one wants the minus character to be part of the set, place it first in the set.

10.3 Repetitions

To create regex that match *strings* we must master so called *repetitions*.

In regex the characters `+` and `*` have a special meaning. The `+`-character indicates that the previous character can occurr once or more times. The regex `"a+"` will thus match the strings *a*, *aa*, *aaa*, *aaaa*, etc.

It is usually more interesting to use the `+`-character after a user defined set or a special character that represents a set. For instance, the regex `"\d+"` will match strings that consist of one or several digits, i.e. integers. Try

```
re.findall(r"\d+", txt)
```

and you should find all integers in our test string.

Try also

```
re.findall(r"\w+", txt)
```

and you should find all word-like character sequences. That is, character sequences that do not contain separator characters.

The character `*` means *zero* or more repetitions. Say we want to find all *words* that contain *or*. The following should achieve this:

```
re.findall(r"\w*or\w*", txt)
```

The strings we search for shall thus first contain zero or more word-characters, then *o* followed by *r*, and then additionally zero or more word-characters.

10.3.1 Eager and Gready

The regex algorithm tries to find a match *at any cost*. This means if a currently explored potential match fails, it will backtrack and try another potential match. This in order to make sure not to miss out on any matches.

Sometimes there are several, mutually excluding one another, matches. If we want to be able to understand or predict which match the algorithm will find, we must know that the regex-algorithm is *eager* and *gready*:

Eager: It finds the match that starts as early as possible.

Gready: It makes the match as long as possible.

The following regex exemplifies this:

```
re.findall(r"\w*\d", "abc123def456ghi")
```

The regex searches for a string that first contains zero or more characters that can be letters or digits and after this contains one digit. There are many substrings that would match such a description, but the regex-algorithm decides to take the match that starts as early as possible (*eager*) and then stops as late as possible (*gready*). It decides for the substring

```
'abc123def456'
```

A ?-character after the star makes the algorithm *lazy* instead of *gready*. The strings that match now still start as early as possible, but are not longer than necessary:

```
re.findall(r"\w*?\d", "abc123def456ghi")
```

will find

```
['abc1', '2', '3', 'def4', '5', '6']
```

10.3.2 Warm up: extract email addresses

We define a new test string that contains two valid email addresses and some more stuff:

```
mtxt = "jox r.nohre@jth.hj.se, bjox@se, adam@example.com, jox@jox@jox.com."
```

Our task shall be to write a regex that finds the two valid email addresses, and only those.

Since we experiment, our first try could be to find words that contain @ ('at'):

```
re.findall(r"\w+@\w+", mtxt)
```

The result should be

```
['nohre@jth', 'bjox@se', 'adam@example', 'jox@jox']
```

which is not what we want, yet.

Task 1

Improve the above regex such that it finds the valid email addresses and no invalid addresses. That is, it should find *r.nohre@jth.hj.se* and *adam@example.com*, but not bjox@se (because the domain between 'at' and 'se' is missing) and neither jox@jox@jox.com (because it contains two 'at').

Hint You probably need parentheses to allow repetitions of certain parts of your regex. When you do this, the output of findall() might be confusing. The reason is that as soon as you use parentheses, you create a *group*, and findall() will focus on delivering the group's contents, instead of the entire match. You can solve this by either adding parentheses around your entire expression (creating an additional giant group that findall() will thus output) or you use *non-capturing* groups. A non-capturing group starts with `(?:` instead of just `(`. Non-capturing groups are not output by findall(). An example illustrates this:

```
>>> re.findall(r'\w+(\.\w+)*@\w+', mtxt)          # capturing group
['.nohre', '', '', '']

>>> re.findall(r'(\w+(\.\w+)*@\w+)', mtxt)        # additional giant group
[('r.nohre@jth', '.nohre'), ('bjox@se', ''), ('adam@example', ''), ('jox@jox', '')]

>>> re.findall(r'\w+(?:\.\w+)*@\w+', mtxt)        # non-capturing group
[('r.nohre@jth', 'bjox@se', 'adam@example', 'jox@jox']
```

Task 2

Does your regex from Task 1 find the email-address jox@jox.com? If yes, how can you avoid this?

Hint: capturing groups...

10.4 Final preparations

Define the following example text:

```
 htmltxt = """ bla bla bla
               <h1>   My Rubric       </h1>
               bla bla bla. """
```

We want now a regex that finds the rubric's text, that is, the text *My Rubric*. With

```
re.findall(r"<h1>\s*.*\s*</h1>", htmltxt)
```

we get the entire h1-tag. If we add a capturing group around `.*`, we only get the text between `<h1>` and `</h1>`.

```
re.findall(r"<h1>\s*(.*)\s*</h1>", htmltxt)
```

However, we get undesired spaces at the end of our desired text:

```
    'My Rubric       '
```

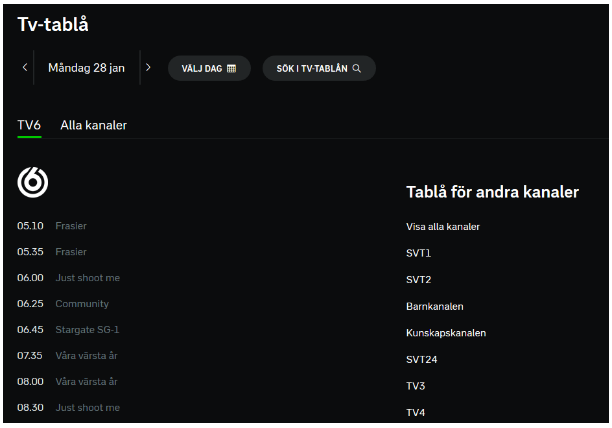
This means, the spaces between "My Rubric" and "</h1>" also come into our group. Why is that? Because * is *gready*! We can avoid the spaces by making it *lazy*:

```
re.findall(r"<h1>\s*(.*?)\s*</h1>", htmltxt)
```

Note finally that one can create multiple groups and each one can be defined as either capturing or non-capturing. This can be useful for the following main task.

10.5 Main task: The Simpsons

On SVT's site you find, among other things, the TV-table for tv6[1], and on a normal day the TV-series *The Simpsons* is emitted several times. We want to write a regex that finds all program points that regard this Simpson series!



05.10	Frasier
05.35	Frasier
06.00	Just shoot me
06.25	Community
06.45	Stargate SG-1
07.35	Våra värsta år
08.00	Våra värsta år
08.30	Just shoot me

[1] https://www.svtplay.se/kanaler?channel=tv6&date=2019-01-28

On Canvas you find the file *tabla.html*. Download it to your folder and enter the following python code in interactive mode:

```
f = open("tabla.html", encoding="utf-8")
txt = f.read()
txt
```

The code puts the entire html-file in the variable `txt` and prints it. Below is an excerpt of that html-file:

```
</div>
</td>
</tr>
<tr class="svtTablaPast svtOpacity-60 tv6" >
<td class="svtTablaTime">
15.00
</td>
<td class="svtJsTablaShowInfo">
<h4 class="svtLink-hover svtTablaHeading">
Simpsons
</h4>
<div class="svtJsStopPropagation">
<div class="svtTablaTitleInfo svtHide-Js">
<div class="svtTablaContent-Description">
<p class="svtXMargin-Bottom-10px">
Amerikansk animerad komediserie från 2007. Säsong 19. Del 9 av 20. En
morgon när  Homer vaknar befinner han sig utomhus och är täckt av snö.
När han kommer hem  är hela familjen borta. Regi: Neil Affleck och
Bob Anderson. Simpsons bor i Springfield.
</p>
</div>
```

A good start can be to find all time points where a tv-program starts. By studying the above html-code we see that a time point typically is given the following way:

```
<td class="svtTablaTime">
15.00
</td>
```

The following regex should match such a time point, and since we put the concrete time into a capturing group it should be output. Test it:

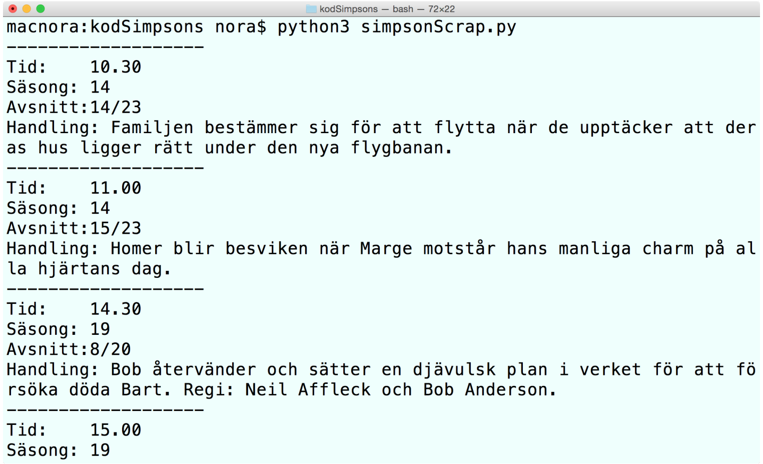
```
re.findall(r'<td class="svtTablaTime">\s*(\d+\.\d+)\s*</td>', txt)
```

After such a time point, according to the above html-code, should come a td-tag (`<td.*?>`), an h4-tag (`<h4.*?>`) followed by the word *Simpsons*, and then </h4>. Between all those tags blanks are allowed (`\s*`).
Therefore, the following regex should find the desired Simpson time points (without the line breaks!):

```
re.findall(r'<td class="svtTablaTime">\s*(\d+\.\d+)\s*</td>
                       \s*<td.*?>\s*<h4.*?>\s*
                       Simpsons
                       \s*</h4>', txt )
```

10.5.1 Task 3

Your task is now to write a short python program that reads the html-file *tabla.html*, extracts all relevant information about all Simpson emissions and presents it in an appropriate way. For instance as in the following screenshot.



```
macnora:kodSimpsons nora$ python3 simpsonScrap.py
-------------------
Tid:    10.30
Säsong: 14
Avsnitt:14/23
Handling: Familjen bestämmer sig för att flytta när de upptäcker att der
as hus ligger rätt under den nya flygbanan.
-------------------
Tid:    11.00
Säsong: 14
Avsnitt:15/23
Handling: Homer blir besviken när Marge motstår hans manliga charm på al
la hjärtans dag.
-------------------
Tid:    14.30
Säsong: 19
Avsnitt:8/20
Handling: Bob återvänder och sätter en djävulsk plan i verket för att fö
rsöka döda Bart. Regi: Neil Affleck och Bob Anderson.
-------------------
Tid:    15.00
Säsong: 19
```

Hints

1. Open the html-code in a text editor and search for an appropriate piece of html-code that describes a Simpson emission plus some other html around it.

2. Open one of the following sites in a webbrowser: *regex101.com*, *regexpal.com*, *regexr.com*.

3. Paste the html-code from 1. into the text field.

4. In the web-site's regex-field: write a regex that captures the relevant html-text and places the desired parts in different capturing groups.

5. Then copy-paste your regex from the webbrowser into your python program.

10.6 Examination

Submit on Canvas your python program and your result from task 3 (e.g. as a comment). Present your solutions to Tasks 1, 2 and 3 to your lab assistant.