Contents

Part I

Introduction to Formal Languages and Translators

Lab 1

Regular Expressions in Python

Goals

In this lab you will learn:

1. What **R**egular **E**xpressions are
2. What are some common **M**eta **C**haracters
3. How to define and use **R**egular **E**xpressions in **P**ython
4. To match custom patterns in various texts
5. To create basic parsers using **R**egular **E**xpressions

Resources

Table 1.1: Lab Resources

Resource	Link
Lab Notebook	`https://colab.research.google.com/drive/16Vgns93LBiY-dAMZonq_ChpEMsvysXRW?usp=sharing`
The utility of regular expressions	`https://levelup.gitconnected.com/why-should-you-learn-regex-66586ba259e0`
Regexes: The Bad, the Better, and the Best	`https://www.loggly.com/blog/regexes-the-bad-better-best/`
Regex 101	`https://regex101.com/`

1.1 History of Regular Expressions

In 1951, mathematician Stephen Cole Kleene described the concept of a regular language, a language that is recognizable by a finite automaton and formally expressible using regular expressions. In the mid-1960s, computer science pioneer Ken Thompson, one of the original designers of Unix, implemented pattern matching in the QED text editor using Kleene's notation.

Since then, regexes have appeared in many programming languages, editors, and other tools as a means of determining whether a string matches a specified pattern. Python, Java, and Perl all support regex functionality, as do most Unix tools and many text editors.

A couple of things to note before we get into the details:

1. Almost every programming language comes with a regular expression library
2. Regexes can be viewed as a standalone topic
3. Such expressions allow us to search for specific patterns in text

1.2 Getting started with Regular Expressions in Python

1.2.1 The *re* module

Regex functionality in Python resides in a module named re. The re module contains many useful functions and methods, most of which you'll learn about in the remainder of this laboratory.

To use the re module in Python we need to import it using `import re`.

Before writing our first regular expression, we need to understand the concept of raw string, because we are going to be looking at raw strings to match them against our regular expressions.

A raw string in Python is just a string prefixed with *r*. This tells Python not to handle backslashes in a special way, since these would normally be used with newlines, tabs etc (e.g. `nt`). To understand this better, try running the following in a Python notebook cell.

Listing 1.2.1: RegEx.ipynb — Python code

```
print('\tTab')
```

You will now see that running the code will result in printing a tab before the text, such as:

Output

```
    Tab
```

Try adding an *r* before the string, such as:

Listing 1.2.2: RegEx.ipynb — Python code

```
print(r'\tTab')
```

You should now see that the output now will print the actual contents of the string, including `\t`.

Output

```
\tTab
```

Since we want our regular expressions to interpret the strings we pass to them as they are, and not have Python perform any processing beforehand, we only use raw strings when defining our expressions.

Concept 1.2.1: Raw Strings

Raw Strings prevent the compiler from handling any special characters contained inside, such as `\n`, `\t`, `\w`.

Let's move on to writing our own regular expressions.

1.2.2 Diving into Regular Expressions

The first notion we need to be aware of when using regular expressions in Python, is that of compiling. The `compile` method allows us to separate patterns into variables, so that we can reuse those for performing multiple searches. Let's see it in action.

Listing 1.2.3: RegEx.ipynb — Python code

```
pattern = re.compile(r'abc')
```

Now that we have written our first pattern, we can find text that matches using the pattern. For searching, we can use the `finditer` method over a variable `text_to_search` containing the text we want to perfrom the search on.

Listing 1.2.4: RegEx.ipynb — Python code

```
pattern = re.compile(r'abc')

matches = pattern.finditer(text_to_search)

for match in matches:
  print(match)
```

If we save this and run it, we see that the output is the following.

Output

```
<re.Match object; span=(1,4), match='abc'>
```

This method returns an iterator that returns all the matches. Each of the match objects show us the span, and the match. The span represents the beginning and end index of the match. When we search the text with the pattern, it found only one match of 'abc', and it found it at indexes 1 to 4 in the original string.

Now if we want to validate this, we could write down the below.

Listing 1.2.5: RegEx.ipynb — Python code

```
print(text_to_search[1:4])
```

```
Output

abc
```

1.2.3 Functions in the *re* module

Python offers two different primitive operations based on regular expressions: `re.match()` checks for a match only at the beginning of the string, while `re.search()` checks for a match anywhere in the string.

```
Output

> print(re.match("c", "abcdef"))
None
> print(re.search("c", "abcdef"))
<re.Match object; span=(2, 3), match='c'>
```

Regular expressions beginning with a caret can be used with search() to restrict the match at the beginning of the string:

```
Output

> print(re.match("c", "abcdef"))
None
> print(re.search("^c", "abcdef"))
None
> print(re.search("^a", "abcdef"))
<re.Match object; span=(0, 1), match='a'>
```

Note 1.2.1

Keep in mind that match and search provide different results! Don't try to use match to find matches at any position in a string.

The functions above can be used directly from the *re* module, but also directly on a pre-compiled regex. Compiling regexes in advance improves code readability and error rates by removing repetition.

```
Output

> pattern = re.compile("c")
> match = pattern.search("abcdef")
```

Note 1.2.2

When looping over multiple strings to check for matches, it's always recommended, for performance reasons, that you compile your regular expression beforehand, rather than directly passing the pattern to the match/search function calls.

Besides search and match, the `re` module also suppports `finditer` and `findall` functions. The former returns an iterator over all substrings that matched the regex from a given string, while the latter returns all matched substrings, grouped.

Note 1.2.3

Find and search functions may seem similar by now. The difference is that `re.search()` stops after the first found occurence, while `re.findall()` and `re.finditer()` scan the string left to right and return all occurences in the found order.

Let's look at the text below to grasp the concepts that will follow. Our `text_to_search` variable now contains the following text:

Listing 1.2.6: RegEx.ipynb — Python code

```
text_to_search = '''
  abcdefghijklmnopqurtuvwxyz
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  1234567890
  Ha HaHa
  MetaCharacters (Need to be escaped):
  . ^ * + ? { } [ ] \ | ( )
  coreyms.com
  321-555-4321
  123.555.1234
  123*555*1234
  800-555-1234
  900-555-1234
  Mr. Schafer
  Mr Smith
  Ms Davis
  Mrs. Robinson
  Mr. T
  '''
```

We have seen that our pattern has matched the 'abc' elements in the first line of the text, but it didn't match the 'ABC' substring in the second line. Intuitively, you may guess that the reason for that relates to case sensitivity. You may also realize by now that the order of the characters specified in our 'abc' expression is also important. If we were to use 'cba' as a pattern, we would get no matches.

1.3 Meta Characters

If you look at the example string that was given, you will see a section 'MetaCharacters' listing some characters that when used inside a regular expression would behave differently. Hence, these would need to be escaped. Let's try some searches to better understand what each meta character does.

If you change the pattern passed to the `compile` method to `r'.'` and run the cell, you will see quite an unexpected output. This is because the *'.'* is a special character in regular expressions used for matching any character. If we wated to actually search for a dot in the text, we would have to escape it in our pattern, so instead write `r'\.'`

So far, we have covered literal searches, which we would already know how to implement without the use of regular expressions. To make things more interesting, we will move on to searching for patterns. To do that, let's look at what each meta character does in a regular expression.

Concept 1.3.1: Raw Strings

Meta Characters are characters that, similar to **keywords**, have a special meaning to a compiler.

Table 1.2: General Meta Characters

Usage	Utility
`.`	Any Character Except New Line
`\d`	Digit (0-9)
`\D`	Not a Digit (0-9)
`\w`	Word Character (a-z, A-Z, 0-9, _)
`\W`	Not a Word Character
`\s`	Whitespace (space, tab, newline)
`\S`	Not Whitespace (space, tab, newline)

Table 1.3: Anchor Meta Characters

Usage	Utility
`\b`	Word Boundary
`\B`	Not a Word Boundary
`^`	Beginning of a String
`$`	End of a String

Table 1.4: Grouping Meta Characters

Usage	Utility
`[]`	Matches Characters in brackets
`[^ ]`	Matches Characters NOT in brackets
`|`	Either Or
`( )`	Group

Table 1.5: Quantifier Meta Characters

Usage	Utility
*	0 or More
+	1 or More
?	0 or One
{3}	Exact Number
{3,4}	Range of Numbers (Minimum, Maximum)

Regex Sample

```
[a-zA-Z0-9\_\.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-\.]+
```

Exercise 1.3.1 *

Look at these patterns in the Colaboratory project:

Listing 1.3.1: Python code

```
pattern1 = re.compile(r'\d')
pattern2 = re.compile(r'\bHa')
pattern3 = re.compile(r'\BHa')

pattern4 = re.compile(r'^Start')
pattern5 = re.compile(r'sentence')
pattern6 = re.compile(r'^sentence')
```

What matches is the program finding for each pattern? Try to guess some matches before looking at the cell outputs.

1.4 Practical examples

We will now write some regular expressions for matching phone numbers. If you look at the phone numbers inside the `text_to_search` string, you will see that we are not able to use string literals to match these numbers because they contain different characters. We will use meta characters to match these.

We are going to first create a pattern for matching phone numbers. We know we can match any digit with `\d`, so we can use this as many times as it is needed to match the beginning of our phone number. After the first section of digits we have dashes or dots, so we could use a '.' to match these. The remaining characters in the phone numbers share the same structure. The final pattern would look like this:

Listing 1.4.1: RegEx.ipynb Python code

```
pattern = re.compile(r'\d\d\d.\d\d\d.\d\d\d\d')
```

In the Colaboratory project you will see a code cell that contains a text sample called data with a series of users of a system, described by their name, phone number, address and emails. We will use this file to validate that our phone number pattern correctly matches substrings from this file.

Listing 1.4.2: RegEx.ipynb — Python code

```
pattern = re.compile(r'\d\d\d.\d\d\d.\d\d\d\d')
matches = pattern.finditer(data)

for match in matches:
  print(matches)
```

If you run this cell, the output should contain multiple results like the one below:

Output

```
<re.Match object; span=7607, 7619), match='911-555-7535'>
```

By now it should be clear that these regular expressions are very useful in parsing specific data in texts.

In the example above we used a dot for matching dashes and dots for phone numbers, but because of that, some numbers may be incorrectly identified since the dot matches any character. For instance, '111*222*1234' would be matched against our pattern. To make our pattern more restrictive and not allow such examples to be matched, we can use the character sets. These are specified in square brackets `[]` containing specific characters to match against. For our example, we could use:

Listing 1.4.3: RegEx.ipynb — Python code

```
pattern = re.compile(r'\d\d\d[-.]\d\d\d[-.]\d\d\d\d')
```

Let's say we would like to only recognize phone numbers for which the first three digits match 800 or 900. We could use the expression below.

Listing 1.4.4: RegEx.ipynb — Python code

```
pattern = re.compile(r'[89]00[-.]\d\d\d[-.]\d\d\d\d')
```

To complicate things a bit, if we wanted to only recognize phone numbers that have the first digit between 1-5, we could continue using character sets, but we would need to add a range inside it, as such:

Listing 1.4.5: RegEx.ipynb — Python code

```
pattern = re.compile(r'[1-5]\d\d[-.]\d\d\d[-.]\d\d\d\d')
```

Ranges are specified using a dash. The dash is a special character inside the character set. If we wanted to match only one lowercase letter, we could use `r'[a-z]'`. Similarily, for both lowercase and uppercase letter, we could use `r'[a-zA-Z]'`.

Another special character in a character set is the caret `^`. While normally, when used as a literal, this would match the beginning of an input, in the character set, the caret *negates* whatever has been specified in the set. If we wanted to recognize any character *except for* letters, we could use `r'[^a-zA-Z]'`.

Let's say we had a text file where we would like to recognize all words made out of three characters such as 'cat', 'sat' etc, but not recognize the word 'bat'. We could use the below regular expression for this.

Listing 1.4.6: RegEx.ipynb Python code

```
pattern = re.compile(r'[^b]at')
```

If we were to have phone numbers containing multiple dashes between the digits, such as '111–222–1234', the regular expression defined above would not match. A character set by itself would always match a single character in the input text. To allow multiple occurences of characters specified in the character set, we can use the quantifier expressions. A useful example for using quantifiers is the following.

If we want to match exactly 3 digits in the phone number, it's not necessary to duplicate the `\d` expression. You can imagine that for complex texts to be parsed, our regular expresions would get quite long and prone to mistakes. To avoid that, we can use the exact number quantifier `{N}`, where N specifies the count.

We could write our initial pattern in a simpler manner, specifying that we want to have exactly N characters of type digit in a group:

Listing 1.4.7: RegEx.ipynb Python code

```
pattern = re.compile(r'\d{3}.\d{3}.\d{4}')
```

Exercise 1.4.1 *

In our `text_to_search` you can see we have some names prefixed by a title such as Mr, Mrs, Ms, Mr. etc. Write a regular expression that would match all of these examples present in the text. Keep an eye out for optional punctuation. It might be useful to review all the meta characters.

1.5 Practice problems

To review what you have learned so far, try solving the exercises below.

Exercise 1.5.1 *

Go to the emails cell in the notebook. There are three email addresses that are fairly different. Try writing a regular expression that would match all these emails.

Hints:
Try to match everything left of the @, by first identifying what characters are present there. After that, move on to the contents of the domain, between the @ and the '.', and finally write down an expression for the ending.
It might be a good idea to start writing an expression that would match the first address, and then update it so that it matches the others as well.

Exercise 1.5.2 **

You should by now be able to write down your own regular expression. However, you may still find it difficult to understand what some pre-written regular expressions are used for. What would you say the below regular expression was written for?

Listing 1.5.4: Python code

```
pattern = re.compile(r'^((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\.){3}
    (25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])$')
```

Question 1.5.1 **

Using `https://regex101.com/` try to identify the flaws in the following regexes and correct them.

1. A regular expression for matching any string containing the letters a, b, c that starts with the character a.

   Listing 1.5.5: Python code

   ```
   pattern = re.compile(r'a(a|b|c)+')
   matches = pattern.findall(str)
   ```

2. A regular expression for matching "aa" or "bb", that would return exactly two matches for the given string.

   Listing 1.5.6: Python code

   ```
   str = '''
       aa
       bb
       123
       not this
       '''
   pattern = re.compile(r'(aa|bb)*')
   matches = pattern.findall(str)
   ```

3. A regular expression for tuples containing single-letter values, such as "(a, b)".

   Listing 1.5.7: Python code

   ```
   pattern = re.compile(r'([a-z],[a-z])')
   matches = pattern.findall(str)
   ```