Regular Expressions

March 19, 2021

Last time looked at the CRISP-DM reference model for approaching data analysis problems, and dicussed some key functionalities of popular and important data science libraries.

Today we will look at something with a quite geeky reputation (https://xkcd.com/208/): regular expressions. They can be very useful for finding patterns in text (and that not only in Python programs!), and therefore software developers should know about them.

Next time we will dive deeper into the concepts of object orientation and object-oriented programming in Python.

String Manipulation

Regular expressions allow us to do advanced things with strings. Let's first recap Python's basic string manipulation features. We have already seen, for example, that the "+" operator is also defined for strings, that `isAlpha()` can be used to check if something is an alphabetical character, that the `lower()` method can be applied to strings to turn all contained characters into lower case (if they are not already), and some other string manipulation functionality. In fact, the string class contains several such methods. For example:

`str.capitalize()` capitalizes the first letter and lowercases the rest

`str.find()` returns the first occurrence of a particular substring in the string

`str.join()` concatenates the strings that are returned by an iterable

`str.replace()` replaces a particular substring by another string

`str.split()` splits a string at a given separation character

`str.splitlines()` splits a multiline string into lines

`str.strip()` strips leading and trailing whitespaces from a string

`str.upper()` uppercases all characters in the string

See https://docs.python.org/3/library/stdtypes.html#string-methods (https://docs.python.org/3/library/stdtypes.html#string-methods) for full reference.

The functions above are all defined for normal, plain strings. Especially for functions like find and replace, however, it is in practice however often useful to *"look for something formatted like ..."* instead. For example, imagine you want to find some unknown e-mail address in a long text, or you want to add the country code to everything that looks like a Dutch mobile phone number. This is where regular expressions can help.

Regular Expressions

Regular expressions are strings that describe patterns. They are not only useful for programmers, but also appear in common user software. Wildcards (often * and ?), filter criteria for text search (pattern matching) and the find/replace feature in text editors (many support regular expressions) are some examples you might have come across earlier. With the `re` module Python supports regular expression operations (https://docs.python.org/3/library/re.html (https://docs.python.org/3/library/re.html)).

For instance, the function `re.findall(pattern,string)` can be used to find all occurrences of a pattern in a string and returns them as a list of strings. We will use this function for the examples in the following and discuss some other functions later on.

Literals

The most basic element of regular expressions are the so-called **literals**, that is, individual characters or character sequences that must match literally. For example:

In [1]:
```
import re
text = "How much wood would a woodchuck chuck if a woodchuck could chuck

print(re.findall("w", text))
print(re.findall("wood", text))
```

```
['w', 'w', 'w', 'w', 'w', 'w']
['wood', 'wood', 'wood', 'wood']
```

Meta-Characters and Escaping

Regular expressions are however not only about describing strings literally, but even more so about sets of strings with particular properties. Therefore, characters are needed that are not interpreted literally. **Meta-characters** are such literals that have a special, sometimes context-dependent meaning:

[ ] ( ) { } | ? + - * ^ $ \ .

We will look at their meaning shortly. For now just note that their interpretation as meta-character can be switched off by putting the backslash character '\' in front. This is also called "**escaping**". For example, the regular expression for describing "1+1=2" is "1\+1=2", the regular expression for describing "C:\temp" is "C:\\temp", and the regular expression for describing "wood?" is "wood\?":

In [2]:
```
print(re.findall("wood?", text))
print(re.findall("wood\?", text))
```

```
['wood', 'wood', 'wood', 'wood']
['wood?']
```

Note that the **standard escapes** (such as the newline "\n" and the horizontal tab "\t") can also be

used in regular expressions.

Character Classes

Pairs of box brackets ("[" and "]") are used to indicate **character classes**, that is, sets of characters from which exactly one is chosen when matching the regular expression to a text. For example, the regular expression "gr[ae]y" matches both "gray" and "grey", and "[Ww]ood" matches the word "wood" in upper and lower case alike:

In [3]:
```
print(re.findall("[Ww]ood", text))
```

```
['wood', 'wood', 'wood', 'wood']
```

Negation ("^") can be used to match any character except for those listed. For example, "wo[^o]" matches all character triples that start with "wo" and do not have an "o" at the third position:

In [4]:
```
print(re.findall("wo[^o]", text))
```

```
['wou']
```

Character classes can also use ranges of characters (corresponding to their order in the ASCII table). For example, "[0-9A-F]" will match hexadecimal digits, "[5-9][0-9]" all numbers between 50 and 99, and "[a-mA-M][n-zN-Z]" all pairs of characters where the first is from the first half and the second from the second half of the alphabet:

In [21]:
```
print(re.findall("[a-mA-M][n-zN-Z]", text))
```

```
['Ho', 'mu', 'hu', 'hu', 'hu', 'co', 'hu']
```

Apart from the "^" at the first position in a character class and the "-" used for defining ranges, special characters lose their special meaning within character classes. That is, the expression "[-(+*)^]" will match any of the literal characters "-", "(", "+", "*", ")" and "^". A closing square bracket "]" can be included in a character class by escaping it ("[\]]").

To simplify things in practice a bit, Python includes a number of **predefined character** classes:

```
.  matches any character (except or including the newline, depen
ding on mode)

\d corresponds to [0-9] and matches any digit

\D corresponds to [^0-9] or [^\d] and matches any non-digit

\s matches any whitespace character (including \t, \n, etc.)

\S corresponds to [^\s] and matches any non-whitespace character

\w matches any word characters (that is, characters that can be
 part of a word in any      language, numbers and the underscor
e)

\W corresponds to [^\w] and matches all non-word characters
```

Boundary Matchers

In addition, there are a couple **boundary matchers** that can be used in regular expressions:

```
^  start of a string, or of a line (in multiline mode)

$  end of a string, or of a line (in multiline mode)

\b word boundary

\B non-word boundary

\A beginning of the input string

\Z the end of the input string
```

For example, the regular expression "d\W" can be used to find all occurrences of "d" at the end of a word:

In [22]:
```
print(re.findall("d\W", text))
```

```
['d ', 'd ', 'd ', 'd?']
```

Repetition Qualifiers

To be able to deal with patterns whose exact length is not known in advance, regular expressions comprise **repetition qualifiers**. For example, if the "?" is appended to a character, character class or more complex expression, it denotes that it is optional and will match zero or one occurrences of it:

In [23]:
```
print(re.findall("wood\??", text))
```

```
['wood', 'wood', 'wood', 'wood?']
```

Other repetition qualifiers are:

```
+     one or more repetitions
*     zero or more repetitions
{n}   exactly n repetitions
{n,m} min n and max m repetitions
{n,}  at least n repetitions
```

For example, we can use it to find all 3-5 character words in the text:

In [5]:
```
print(re.findall("\w{3,5}\W", text))
```

```
['How ', 'much ', 'wood ', 'would ', 'chuck ', 'chuck ', 'chuck ', 'cou
ld ', 'chuck ', 'wood?']
```

By default the repetitions quantifiers are "greedy", that is, they match as much of the input text as possible. To make repetition quantification non-greedy ("reluctant"), a "?" can simply be appended, causing the regular expression engine to try to match the expression using as few input text as possible. For example, the expression "A.*a" matches "Anna-Lena" completely, while "A.*?a" matches only "Anna". The following example illustrates the difference on our input text:

In [6]:
```
print(re.findall("\w+", text))
print(re.findall("\w+?", text))
```

```
['How', 'much', 'wood', 'would', 'a', 'woodchuck', 'chuck', 'if', 'a',
'woodchuck', 'could', 'chuck', 'wood']
['H', 'o', 'w', 'm', 'u', 'c', 'h', 'w', 'o', 'o', 'd', 'w', 'o', 'u',
'l', 'd', 'a', 'w', 'o', 'o', 'd', 'c', 'h', 'u', 'c', 'k', 'c', 'h',
'u', 'c', 'k', 'i', 'f', 'a', 'w', 'o', 'o', 'd', 'c', 'h', 'u', 'c',
'k', 'c', 'o', 'u', 'l', 'd', 'c', 'h', 'u', 'c', 'k', 'w', 'o', 'o',
'd']
```

Capturing Groups

Round brackets around an expression denote a **capturing group**. That is, the result of matching the regular expression will be what is inside the group. A quantifier following a closing round bracket applies to the whole group. For example, "Uni(versiteit)? Utrecht" matches "Uni Utrecht" and "Universiteit Utrecht", but the returned result of the matching will be "versiteit" and "", respectively. In our running example, we can use it, for example, to find all words preceding an occurrence of "wood":

In [7]:
```
print(re.findall("(\w+)\Wwood", text))
```

```
['much', 'a', 'a', 'chuck']
```

This shows that capturing groups are useful to find specific, varying text that comes together with specific, fixed pieces of text. Another, maybe more practical example in this regard is to find the text between particular tags in a HTML or XML file:

In [8]:
```
print(re.findall("<b>(\w+)</b>", \
                 "Bold font marks <b>important</b> words."))
```

```
['important']
```

Capturing groups can also be nested, then returning results for the different capturing levels:

In [28]:
```
print(re.findall("((\w+)\Wwood)", text))
```

```
[('much wood', 'much'), ('a wood', 'a'), ('a wood', 'a'), ('chuck woo
d', 'chuck')]
```

Groups can be denoted as non-capturing by using "(?:" instead of a single round bracket at the beginning. This can be handy to structure more complex regular expressions, but without including all thus defined groups into the returned result. For example:

In [29]:
```
print(re.findall("((?:\w+)\W(?:wood))", text))
```

```
['much wood', 'a wood', 'a wood', 'chuck wood']
```

Another kind of non-capturing groups can be used to configure the behavior of a regular expression more precisely. After opening a group with "(?", one or more flags from the set 'a' (ASCII-only matching), 'i' (ignore case), 'L' (locale-dependent), 'm' (multi-line), 's' (dot matches all), 'u' (Unicode matching), 'x' (verbose) can be set. For example, the ignore-case flag can be used to match the file ending "pdf" to "pdf", "Pdf" and "PDF" alike:

In [30]:
```
print(re.findall("(?i)pdf", "PDF, pdf, or Pdf?"))
```

```
['PDF', 'pdf', 'Pdf']
```

The flags 'i', 's', 'm' and 'x' can also be switched off again with a preceding '-'.

Alternatives

We have seen earlier that character classes can be used to describe alternatives at the character level. With the symbol "|" it is also possible to denote longer more complex expressions as **alternatives**. They are tried from left to right in a non-greedy fashion, that is, once one alternative matches, it is accepted. For example:

In [31]:
```
print(re.findall("wood|chuck|woodchuck", text))
```

```
['wood', 'wood', 'chuck', 'chuck', 'wood', 'chuck', 'chuck', 'wood']
```

Another example (words ending on "od" or "uld"):

In [32]:
```
print(re.findall("(\w+(ood|ould))", text))
```

```
[('wood', 'ood'), ('would', 'ould'), ('wood', 'ood'), ('wood', 'ood'),
('could', 'ould'), ('wood', 'ood')]
```

Or with a non-capturing inner group:

In [33]:
```
print(re.findall("(\w+(?:ood|ould))", text))
```

```
['wood', 'would', 'wood', 'wood', 'could', 'wood']
```

A Realisitic Example

Finally, let's have a look a realistic example. Suppose you have downloaded a plain-text sequence file from the EMBL database, say https://www.uniprot.org/uniprot/P05787.txt (https://www.uniprot.org/uniprot/P05787.txt). We want to retrieve the information from the "RL" lines, that is, the common citation information for the reference, that is usually sufficient to find the papers in question. This can be done in Python as follows:

In [10]:
```
import re

with open("data/P05787.txt", "r") as file:
    dbentry = file.read()
reference_locations = re.findall("(?m)^RL\W+(.+)\.$", dbentry)

print(reference_locations)
```

```
['Gene 86:241-249(1990)', 'Mol. Endocrinol. 4:370-374(1990)', 'New Bio
l. 2:464-478(1990)', 'J. Biol. Chem. 272:7556-7564(1997)', 'Nat. Genet.
36:40-45(2004)', 'Submitted (APR-2005) to the EMBL/GenBank/DDBJ databas
es', 'Nature 440:346-351(2006)', 'Submitted (JUL-2005) to the EMBL/GenB
ank/DDBJ databases', 'Genome Res. 14:2121-2127(2004)', 'Mol. Cell. Bio
l. 9:1553-1565(1989)', 'Differentiation 33:69-85(1986)', 'Electrophores
is 18:605-613(1997)', 'J. Biol. Chem. 267:3901-3906(1992)', 'J. Cell Bi
ol. 117:583-593(1992)', 'Arch. Oral Biol. 45:879-887(2000)', 'J. Biol.
Chem. 275:14910-14915(2000)', 'J. Biol. Chem. 277:10767-10774(2002)',
'J. Biol. Chem. 277:10775-10782(2002)', 'J. Cell Sci. 118:1081-1090(200
5)', 'Mol. Biol. Cell 16:4280-4293(2005)', 'Proteomics 5:2227-2237(200
5)', 'Cell 127:635-648(2006)', 'J. Biol. Chem. 281:16453-16461(2006)',
'Nat. Biotechnol. 24:1285-1292(2006)', 'Mol. Cell 31:438-448(2008)', 'P
roc. Natl. Acad. Sci. U.S.A. 105:10762-10767(2008)', 'Mol. Cell. Biol.
29:1834-1854(2009)', 'Mol. Cell. Proteomics 8:1751-1764(2009)', 'Scienc
e 325:834-840(2009)', 'J. Biol. Chem. 285:34062-34071(2010)', 'Sci. Sig
nal. 3:RA3-RA3(2010)', 'BMC Syst. Biol. 5:17-17(2011)', 'Mol. Cell. Pro
teomics 10:M111.012658.01-M111.012658.12(2011)', 'Mol. Pharmacol. 79:40
0-410(2011)', 'Sci. Signal. 4:RS3-RS3(2011)', 'Biochim. Biophys. Acta 1
820:1839-1848(2012)', 'J. Proteome Res. 12:260-271(2013)', 'J. Proteomi
cs 96:253-262(2014)', 'Mol. Cell. Proteomics 13:372-387(2014)', 'Nat. S
truct. Mol. Biol. 21:927-936(2014)', 'Proc. Natl. Acad. Sci. U.S.A. 11
1:12432-12437(2014)', 'Nat. Struct. Mol. Biol. 24:325-336(2017)', 'Pro
c. Natl. Acad. Sci. U.S.A. 100:6063-6068(2003)']
```

As another example, let's say we want to retrieve the organism classifications, that is, the terms that appear separated by semicolons in the lines starting with "OC". This cannot be done with a single regular expression (because a capturing group will always only return the last occurrence of the corresponding match, but we want to get all), so we need to get the OC lines first end then get the individual organism classifications from there. In Python:

In [11]:
```
oc_lines = re.findall("(?m)^OC\W+(.+)$", dbentry)
organism_classifications = []
for oc_line in oc_lines:
    ocs = re.findall("\w+", oc_line)
    organism_classifications += ocs

print(organism_classifications)
```

```
['Eukaryota', 'Metazoa', 'Chordata', 'Craniata', 'Vertebrata', 'Euteleo
stomi', 'Mammalia', 'Eutheria', 'Euarchontoglires', 'Primates', 'Haplor
rhini', 'Catarrhini', 'Hominidae', 'Homo']
```

For further details on regular expressions (there are a number of things that have not been covered here), see the documentation of the re package at https://docs.python.org/3/library/re.html (https://docs.python.org/3/library/re.html) or the Regular

Expression How To at https://docs.python.org/3/howto/regex.html (https://docs.python.org/3/howto/regex.html).

Advanced String Manipulation with Regular Expressions

As mentioned above, `re.findall()` is not the only useful function when working with regular expressions. Again, the `re` module documentation online contains full reference about the available functions. We will look at a few concrete examples here.

The `re.match()` and `re.fullmatch()` functions can be used to test if a string matches a given pattern. For `match()`, the beginning of the string must match, while for `fullmatch()` the complete string needs to match the pattern. In case of a match, a match object is returned, and None otherwise:

In [12]:
```
import re

pattern_06nummer = "06[1-5][1-9][0-9]{6}"

# function re.match
matcher = re.match(pattern_06nummer, "0635436843")
print(matcher)
matcher = re.match(pattern_06nummer, "0635436843xyz")
print(matcher)

# function re.fullmatch
matcher = re.fullmatch(pattern_06nummer, "0635436843")
print(matcher)
matcher = re.fullmatch(pattern_06nummer, "0635436843xyz")
print(matcher)
```

```
<re.Match object; span=(0, 10), match='0635436843'>
<re.Match object; span=(0, 10), match='0635436843'>
<re.Match object; span=(0, 10), match='0635436843'>
None
```

Obviously, the match object contains information about the match, such as the begin and end index of the match in the string, and the matched sequence. Accordingly, it provides a number of frequently useful functions and attributes (see https://docs.python.org/3/library/re.html#match-objects (https://docs.python.org/3/library/re.html#match-objects)). And it has the truth value True itself (while None has the truth value False), so it can easily be used to check if pattern matches the string:

In [14]:
```
# function re.match
matcher = re.match(pattern_06nummer, "0635436843")
if matcher:
    print("Match found!")
else:
    print("No match found.")

matcher = re.match(pattern_06nummer, "0635436843xyz")
if matcher:
    print("Match found!")
else:
    print("No match found.")

# function re.fullmatch
matcher = re.fullmatch(pattern_06nummer, "0635436843")
if matcher:
    print("Match found!")
else:
    print("No match found.")

matcher = re.fullmatch(pattern_06nummer, "0635436843xyz")
if matcher:
    print("Match found!")
else:
    print("No match found.")
```

```
Match found!
Match found!
Match found!
No match found.
```

The `re.search()` function can be used to find the first (if any) occurrence of a pattern in a string. For example, the first occurrence of "wood", "chuck" or "woodchuck" in the woodchuck tongue twister:

In [15]:
```
matcher = re.search("wood|chuck|woodchuck", text)
print(matcher)
```

```
<re.Match object; span=(9, 13), match='wood'>
```

As a last example, `re.sub()` can be used to replace all occurrences of a pattern in a string by something else. For instance, all occurrences of "wood", "chuck" or "woodchuck" by "gotcha!":

In [16]:
```
newtext = re.sub("wood|chuck|woodchuck", "gotcha!", text)
print(newtext)
```

```
How much gotcha! would a gotcha!gotcha! gotcha! if a gotcha!gotcha! cou
ld gotcha! gotcha!?
```

Exercises

Please use Quarterfall to submit and check your answers.

1. Understanding Regular Expressions

A. What is described by the following regular expressions?

1. `([01]?[0-9]|2[0-3]):[0-5][0-9]`
2. `[^\s]+\.py$`
3. `[^\s]+(\.(?i)(jpg|png|gif|bmp))$`
4. `[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}`
5. `<tag\b[^>]*>(.*?)</tag>`

B. The format of accession numbers in the UniProt database can be described by the regular expression below. What does it mean? Give 3 examples of valid accession numbers.

```
[OPQ][0-9][A-Z0-9]{3}[0-9]|[A-NR-Z][0-9]([A-Z][A-Z0-9]{2}[0-9]){1,2}
```

Answer: UniProt accession numbers can either consist of one of the letters O, P and Q followed by a digit (0-9), followed by three letters or digit, and closing with a digit, or of one of the letters A-N or R-Z followed by a digit, followed by one or two times a letter followed by two letters or digits and one digit.

2. Writing Regular Expressions

Write regular expressions that describe the following:

1. Single-digit and two-digit numbers greater than zero
2. Two-and-more-digit numbers greater than zero
3. Single-and-more-digit numbers greater than zero
4. Four-digit numbers greater than zero
5. Four-and-more digit numbers greater than zero
6. Two-to-four-digit numbers greater than zero
7. Floating-point numbers between 0 and 1 (including)
8. Temperatures (C or F) with 1 decimal place
9. Dutch postal codes
10. BSN numbers

3. Information from a Database Entry

Download the plain UniProt database entry that we already used in the lecture (https://www.uniprot.org/uniprot/P05787.txt (https://www.uniprot.org/uniprot/P05787.txt)). Write a Python program that retrieves and prints the creation and modification dates from the DT lines, and the keywords from the KW lines. You can use the following code to read the content of the file into a string variable dbentry:

```
with open("P05787.txt", "r") as file:
    dbentry = file.read()
```

The output should be something like:

```
['01-NOV-1988', '23-JAN-2007', '13-FEB-2019']
['Acetylation', 'Alternative', 'splicing', 'Coiled', 'coil',
'Complete', 'proteome', 'Cytoplasm', 'Direct', 'protein',
'sequencing', 'Disease', 'mutation', 'Glycoprotein', 'Host', 'vi
rus', 'interaction', 'Intermediate', 'filament', 'Isopeptide',
 'bond', 'Keratin', 'Methylation', 'Nucleus', 'Phosphoprotein',
 'Polymorphism', 'Reference', 'proteome', 'Ubl', 'conjugation']
```

4. String Reformatting

Write a Python program that reads the same input file as in the previous exercise, anonymizes the authors in the RA lines, and prints the modified entry to the screen. The output should be something like:

```
ID   K2C8_HUMAN              Reviewed;         483 AA.
AC   P05787; A8K4H3; B0AZN5; F8VXB4; Q14099; Q14716; Q14717; Q53
GJ0;
AC   Q6DHW5; Q6GMY0; Q6P4C7; Q96J60;
[...]
RP   ARG-47, AND IDENTIFICATION BY MASS SPECTROMETRY [LARGE SCAL
E
RP   ANALYSIS].
RC   TISSUE=Colon carcinoma;
RX   PubMed=24129315; DOI=10.1074/mcp.O113.027870;
RA   (authors anonymized)
RA   (authors anonymized)
RA   (authors anonymized)
RT   "Immunoaffinity enrichment and mass spectrometry analysis o
f protein
RT   methylation.";
[...]
```

In [ ]: