Python Project Course

Day3 Assignment

Research paper on Regular Expressions

By Peng Wang

Instructor: Mr. Vijay Kumar

Metro College of Technology

2021.06.29

Contents

To simply put, Regular Expressions (regex) help us define text patterns. Crafting a regex that matches exactly what you want and nothing more is the goal.

Having the defined text patterns, we can –

- Validate data submitted by users;
- Extract IP address from a long text document;
- Convert a comma-delimited data into JSON format;
- Test if a credit card number has the correct number of digits;
- Check if an email address or a phone number is in a valid format;
- Search programming code;
- …

What are Regular Expressions?

- Regular Expressions, also called Regex [re(d)-djeks] for short.
- It is a tool for searching, matching or replacing parts of a text,
- By setting a set of symbols to describe the patterns that can identify those parts.
- Regex is not a programming language but a formal language that follows a set of rules that computers understand – No variables and don't set values.
- A good Regex should match the text that you want to target and ONLY that text, nothing more – No False Positives to expect.
- Regex can play an important role in the data pre-processing phase, and it helps in manipulating textual data as a prerequisite for data science projects involving text mining.

Ordinary Characters vs. Special Characters

Ordinary Characters or Literal Characters

The simplest Regex – they match themselves EXACTLY.

/c/ matches 'c'

/cat/ matches 'cat'

Special Characters / Metacharacters

- **<u>Wildcard Metacharacter</u> (dot or period)**

  Represents ANY character but line-break.

  /h.t/ matches 'hat', 'h1t', and 'h-t', but not 'heat'.

  It represents ONLY ONE character in the string.

- **\ <u>Escaping Metacharacters</u>: to convert a metacharacter as a literal character.**

  /\./ matches '.'

  /\\/ matches '\'

  /\// matches '/'

/9\.00/ matches ‘9.00’, but not ‘9000’ or ‘9-00’ as it’s not same as /9.00/

Other Special Characters

Spaces: space is a character! /c a t/ matches ‘c a t’, same as /c.a.t/

Tabs (\t): /a\tb/ matches ‘a	b’

Line returns (\r, \n, \r\n)

Character Sets and Range

[] Opening and Closing Square Brackets

Any ONE of several characters within [].

But only ONE character.

Order of characters in the set does NOT matter.

/[aeiou]/ matches any one vowel.

/gr[ea]y/ matches ‘grey’ or ‘gray’ but not ‘greay’.

- Dash

Character Ranges – includes all characters between two characters, inclusively.

/[0-9]/

/[A-Za-z]/

/[50-99]/ matches 5, 0-9, and 9. NOT numbers from 50 till 99.

/[0-9] [0-9] [0-9]- [0-9] [0-9] [0-9]- [0-9] [0-9] [0-9] [0-9]/ matches 555-666-7890,

Although this is not the smart / proper way to extract a phone number.

^ Caret 插入符

Negate a character set.

NOT (exclusive from) any one of several characters.

/[^aeiou]/ matches any non-vowel characters.

/see[^mn]/ matches ‘sees’, ‘see1’, ‘see.’, ‘see ’, but not ‘seem’, ‘seen’, nor ‘see’.

/[^a-zA-Z]/ matches non-letters in both lower and upper cases.

Shorthand Character Sets

Shorthand looks very different from Character Sets, but it represents a Character Set but nothing else, like int for Integer, or ca for Canada.

\d	**Digit**	**[0-9]**

\w	**Word character**	**[a-zA-Z0-9_] include underscore which is a Word character.**
\s	**Whitespace**	**[\t\r\n]**
\D	**NOT Digit**	**[^0-9]**
\W	**NOT Word character**	**[^a-zA-Z0-9]**
\S	**NOT Whitespace**	**[^\t\r\n]**

[^\d] = [\D]

[^\d\s] is NOT [\D\S]

[^\d\s] = NOT digit OR space character

[\D\S] = EITHER NOT digit OR NOT space character – rarely use.

Repetition Metacharacters

* Preceding item, zero or more times

/\d\d\d\d*/ = numbers with three digits or more,

same as

/\d\d\d+/

+ Preceding item, one or more times

/.+/ = any string except a line return

/Good .+\./ = 'Good morning.', 'Good day.'…

/\d+/ = '60823' etc…

/\d\d\d+/ = numbers with three digits or more

/\s[a-z]+ed\s/ = any lowercase words ending in 'ed', with spaces in before and after.

? Preceding item, zero or one time

/colou?r/ = 'color' and 'colour'

/apples*/ = 'apple', 'apples', 'applesssssss'

/apples+/ = 'apples', 'applesssss', but not 'apple'

/apples?/ = 'apple', 'apples', but not 'applesssss'

{} Quantified repetition of the preceding item

{min, max}, min and max are positive,

min is mandatory and can be zero, max is optional.

\d{4,8} = numbers with four to eight digits.

\d{4} = numbers with exactly four digits (min is max).

\d{4,} = numbers with four or more digits (max is infinite).

\d{0,} = \d*

\d{1,} = \d+

/\d{3}-\d{3}-\d{4}/ = US/CAN phone numbers.

Greedy Expressions: /./, /*/, /+/

When we use repetition expressions, strings might have uncertain length and found multiple choices.

For example, /".+", ".+"/ might give us many combinations from strings "Peng", "Wang", "from", "Ontario" , however only my full name is the target that I want.

Which combination will be the default and meet the requirment?

To figure that out we need to understand how the regex engine 'think' about this problem.

The symbol '+', one of the standard repetition quantifiers, is greedy. Same as other repetition metacharacters, they always trying to match the longest possible string(s), by DEFAULT.

For example,

/.*[0-9]+/ matches "Page 266" ➔

.* part did the job of matching "Page 26" ➔

[0-9]+ portion did the matching "6".

This proves that .* matched as much as possible (so greedy), before giving its (tiniest available of) control to the next expression part.

Lazy (or Non-greedy) Expressions: /?/

When talking Lazy Expressions, symbol ? plays the different role as it does as a repetition metacharacter. It instructs the quantifier to play 'lazy' instead of 'greedy'.

Therefore, instead of matching as much as possible, lazy expression will match as LITTLE as possible, before giving contaol to the next expression part.

For the same example from previous greedy section, we add a ? after .* in below:

/.*?[0-9]+/ matches "Page 266" ➔

.*? part did the job of matching "Page " ➔

(it was so 'lazy' therefore quit as soon as the next part can handle the work) ➔

[0-9]+ portion did the matching "266" (no ? mark, so greedy be default, matched all numbers).

If we add a ? after [0-9]+:

/.*?[0-9]+?/ matches “Page 266” ➔

.*? part did the job of matching “Page ” ➔

[0-9]+? portion did the matching “2” ➔

(it is in ‘lazy’ version so it quits as soon as the 1st number being found)

Grouping and Alternation

(): to group a portion of expressions.

$: to call for each group.

|: is an OR operator, match expression on the left or on the right. Leftmost gets precedence.

Eg.:

/w(ei|ie)rd/ = “weird” or “wierd”

Anchors

Anchors reference a position, NOT an actual character as they got Zero-width.

^	Start of string/line
\A	Start of string, never end of line
$	End of string/line
\Z	End of string, never end of line

Eg.: To validate/match an email address:

/^\w+@\w+\.[a-z]{3}$/ = someone@nowhere.com

Signle-line mode

^ and $, \A and \Z, they DO NOT match at line breaks (return key)

Multiline mode

^ and $ will match at the start and end of lines

\A and \Z DO NOT match at line breaks because they represent the beginning of a string.

Word Boundaries

We can anchor regular expressions to word boundaries. That is the start or the end of a word.

\b	indicate a word boundary (anything not a-zA-Z0-9_ is considered a word boundary)
\B	the opposite, something that is NOT a word boundary

For example, to match a word ends with letter s:

We picked apples. ➔

/\b\w+s\b/ will do the job more efficient than /\w+s/, because it immediately eliminated any letters without boundary, and just focused on the letters have boundary.

Word boundaries are a helpful way for us to narrow our regular expressions and to anchor them the same way that we anchored the start and end. They can also make your regular expressions more efficient or faster.

By adding the word character, we narrow down the choices that have to be tried, we've become more specific about what we're looking for, and that enables the regular expression engine to be more efficient.

Regex in Python

Regex can be run on many majority engieens, include Python.

Python has a library specifically handle Regex, named 're'. On top of the general Regular Expression concepts described in above, re contains some very useful functions such as match(), compile(), search(), findall(), sub() for search and replace, and split() or some more.

r - Raw String Literal

```
pattern = r"Cookie"
```

We noticed 'r' before the string. This is how we craft Regex pattern in Python, it ensures the string literal is interpreted and stored as they appear. For example, r\ will prevent '\' being recognized as escape, if we need to treat '\' as a regular character in the text.

Re.Match()

This function checks for a match only at the beginning of the string, and returns a match object if the string matches the regex pattern. Otherwise it will return NONE.

Re.Search() and Group()

This pair of functions used together quite often.

Re.Search() scans through the given string, locate the first place where matches the regex pattern. The difference from Match() is that Match() focus on the BEGINNING of the string, whereas Search() forcus on the FIRST match.

Group() returns the string matched through the Search() function.

```
re.search(r'Co.k.e', 'Cookie').group()
```

```
'Cookie'
```

Compile(pattern, flags=0)

Compile() function can SAVE the regex object for reuse purpose, other then repeatedly writing regex pattern every where, to improve the efficiency.

```
pattern = re.compile(r"cookie")
sequence = "Cake and cookie"
pattern.search(sequence).group()
```

```
'cookie'
```

Findall(pattern, string, flag=0)

This function returns all the matchings as a list of strings.

```
statement = "Please contact us at: support@datacamp.com, xyz@datacamp.com"

#'addresses' is a list that stores all the possible match
addresses = re.findall(r'[\w\.-]+@[\w\.-]+', statement)
for address in addresses:
    print(address)
```

```
support@datacamp.com
xyz@datacamp.com
```

[The End]

References:

https://www.linkedin.com/learning/learning-regular-expressions-2/write-text-matching-patterns

https://www.datacamp.com/community/tutorials/python-regular-expression-tutorial#table

https://docs.python.org/3/library/re.html