Part 13: Step by Step Guide to Master NLP – Regular Expressions

CHIRAG GOYAL — June 25, 2021

Advanced NLP Python Text Unstructured Data

This article was published as a part of the Data Science Blogathon

Introduction

This article is part of an ongoing blog series on Natural Language Processing (NLP). From this article, we will start our discussion on Regular Expressions. When a data scientist comes across a text processing problem whether it is searching for titles in names or dates of birth in a dataset, regular expressions rear their ugly head very frequently.

They form part of the basic techniques in NLP and learning them will make you a more efficient programmer. Therefore, Regular Expression is one of the key concepts of Natural Language Processing that every NLP expert should be proficient in.

So, In this article, we will see all the necessary concepts related to Regular Expressions and also discuss important functionalities related to Regular Expressions.

This is part-13 of the blog series on the Step by Step Guide to Natural Language Processing.

Table of Contents

Regular expressions or RegEx is defined as a sequence of characters that are mainly used to find or replace patterns present in the text. In simple words, we can say that a regular expression is a set of characters or a pattern that is used to find substrings in a given string.

A regular expression (RE) is a language for specifying text search strings. It helps us to match or extract other strings or sets of strings, with the help of a specialized syntax present in a pattern.

For Example, extracting all hashtags from a tweet, getting email iD or phone numbers, etc from large unstructured text content.

Sometimes, we want to identify the different components of an email address.

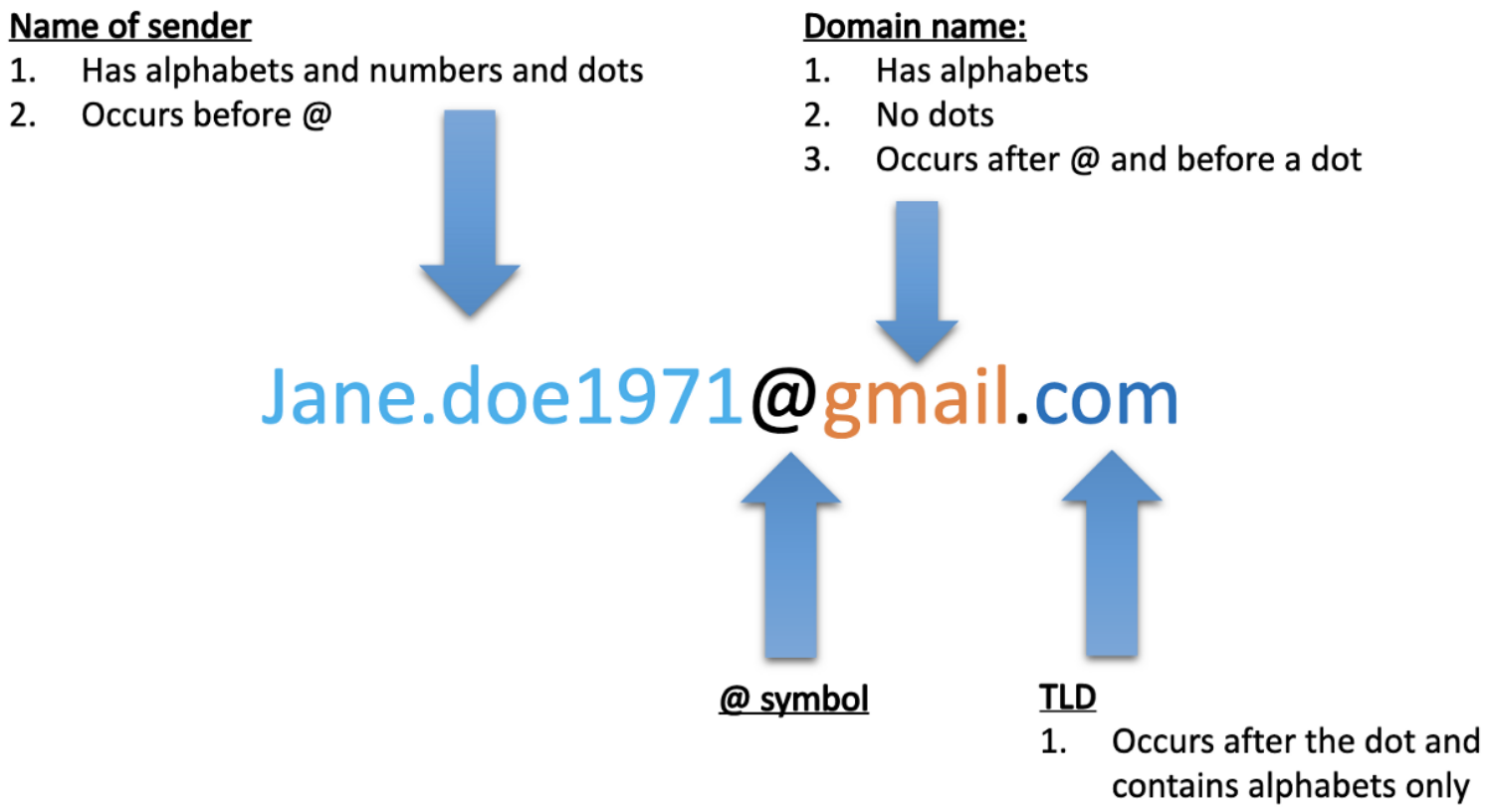

Image Source: Google Images

Simply put, a regular expression is defined as an ”instruction” that is given to a function on what and how to match, search or replace a set of strings.

Regular Expressions are used in various tasks such as,

- Data pre-processing,
- Rule-based information Mining systems,
- Pattern Matching,
- Text feature Engineering,
- Web scraping,
- Data Extraction, etc.

How do Regular Expressions Work?

Let’s understand the working of Regular expressions with the help of an example:

Consider the following list of some students of a School,

```
Names: Sunil, Shyam, Ankit, Surjeet, Sumit, Subhi, Surbhi, Siddharth, Sujan
```

And our goal is to select only those names from the above list which match a certain pattern such as something like this **S u _ _ _**

The names having the first two letters as S and u, followed by only three positions that can be taken up by any of the letters

Let's go one by one, the name **Sunil**, **Sumit,** and **Sujan** fit this criterion as they have S and u in the beginning and three more letters after that. While the rest of the three names are not following the given criteria. So, the new list extracted is given by,

```
Extracted Names: Sunil, Sumit, Sujan
```

What exactly we have done here is that we have a pattern and a list of student names and we have to find the name that matches the given pattern. That's exactly how regular expressions work.

In RegEx, we have different types of patterns to recognize different strings of characters.

Properties of Regular Expressions

Some of the important properties of Regular Expressions are as follows:

1. The Regular Expression language is formalized by an American Mathematician named Stephen Cole Kleene.

2. Regular Expression(RE) is a formula in a special language, which can be used for specifying simple classes of strings, a sequence of symbols. In simple words, we can say that Regular Expression is an algebraic notation for characterizing a set of strings.

3. Regular expression requires two things, one is the pattern that we want to search and the other is a corpus of text or a string from which we need to search the pattern.

Mathematically, we can define the concept of Regular Expression in the following manner:

1. ε is a Regular Expression, which indicates that the language is having an empty string.

2. φ is a Regular Expression which denotes that it is an empty language.

3. If X and Y are Regular Expressions, then the following expressions are also regular.

- X, Y
- X.Y(Concatenation of XY)
- X+Y (Union of X and Y)
- X*, Y* (Kleen Closure of X and Y)

4. If a string is derived from the above rules then that would also be a regular expression.

How can Regular Expressions be used in NLP?

In NLP, we can use Regular expressions at many places such as,

1. To Validate data fields.

For Example, dates, email address, URLs, abbreviations, etc.

2. To Filter a particular text from the whole corpus.

For Example, spam, disallowed websites, etc.

3. To Identify particular strings in a text.

For Example, token boundaries

4. To convert the output of one processing component into the format required for a second component.

Now let's discuss a crucial concept which you must know while studying Regular Expressions. On the basis of your prior knowledge of Python Programming language, you might know that Python raw string treats backslash() as a literal character.

To understand this thing, let's look at the following example:

In the following example, we have a couple of backslashes present in the string. But in string python treats n as "move to a new line".

```
# normal string
path = "C:\desktop\nayan"
print("string:",path)
```

```
string: C:\desktop
ayan
```

After seeing the output, you can observe that n has moved the text after it to a new line. Here "nayan" has become "ayan" and n disappeared from the path. This is not what we want.

So, to resolve this issue, now we use the "r" expression to create a raw string:

```
#raw string
path = r"C:\desktop\nayan"
print("raw string:",path)
```

```
raw string: C:\desktop\nayan
```

Again after seeing the output, we have observed that now the entire path has printed out here by simply using "r" in front of the path.

Therefore, It is always recommended to use raw string instead of normal string while dealing with Regular expressions.

Common Regex Functions used in NLP

To work with Regular Expressions, Python has a built-in module known as "re". Some common functions from this module are as follows:

- re.search()
- re.match()
- re.sub()
- re.compile()
- re.findall()

Let us look at each function with the help of an example:

re. search( )

This function helps us to detect whether the given regular expression pattern is present in the given input string. It matches the first occurrence of a pattern in the entire string and not just at the beginning. It returns a Regex Object if the pattern is found in the string, else it returns a None object.

```
Syntax: re.search(patterns, string)
```

Within search functions, we can use different flags to perform specific operations. **For Example,**

```
‘re.I’ – to ignore the case (either uppercase or lowercase) of the text
're.M' – enables to search the string in multiple lines
re.search(pattern, string, flags=re.I | re.M)
```

In the following example, we search for the pattern "founder" in a given string

```
import re
result = re.search('Founder',r'Kunal Jain is the CEO and Founder of Analytics Vidhya')
print(result.group())
```

```
Founder
```

re.match( )

This function will only match the string if the pattern is present at the very start of the string.

```
Syntax: re.match(patterns, string)
```

In the following example, we match a word at the beginning of the string:

```
import re
result = re.match('Analytics',r'Analytics Vidhya is the largest data Scientists community')
print(result)
print(result.group())
```

```
<re.Match object; span=(0, 9), match='Analytics'>
Analytics
```

Here **Pattern** = 'Analytics' and **String** = 'Analytics Vidhya is the largest data Scientists community'. Since the required pattern is present at the beginning of the string we got the matching Object as an output. And we know that the output of the re.match is an object, so to get the matched expression, we will use the **group()** function of the match object.

As we can observe that we have got our required output using the group() function. Now let us have a look at the other example which presents the second case:

```
import re
result = re.match('Scientists',r'Analytics Vidhya is the largest data Scientists community')
print(result)
```

```
None
```

Here as you can notice, our pattern(Scientists) is not present at the beginning of the string, hence we got **None** as our output.

re.sub( )

This function is used to substitute a substring with another substring.

```
Syntax: re.sub(patterns, Substitute, Input text)
```

In the following example, we replace xxx and yyy with abc in the given string:

```
import re
string = "abc xxx abc yyy"
new_string = re.sub(r"xxx|yyy", "abc", string)
print(new_string)
```

```
abc abc abc abc
```

re.compile( )

This function stores the regular expression pattern in the cache memory for faster searches. So, we have to pass the regex pattern to re.compile() function.

```
Syntax: re.compile(patterns, repl, string)
```

```
import re
pattern=re.compile('Analytics')
result1=pattern.findall('Analytics Vidhya teaches Analytics')
print(result1)
result2=pattern.findall('Analytics Vidhya is most popular website for Data Science')
print(result2)
```

```
['Analytics', 'Analytics']
['Analytics']
```

re.findall( )

This function will return all the occurrences of the pattern from the string. I would recommend you to always use re.findall(). It can work like both re.search() and re.match(). Therefore, the result of the findall() function is a list of all the matches

```
Syntax: re.findall(patterns, string)
```

```
import re
result = re.findall('founded',r'Andrew NG founded Coursera. He also founded deeplearning.ai')
print(result)
```

```
['founded', 'founded']
```

Basics in Regular Expressions

Let's now discuss some of the basics used in Regular Expressions:

Brackets ([ ])

They are used to specify a disjunction of characters.

For Examples,

```
/[cC]hirag/ → Chirag or chirag
/[xyz]/ → 'x', 'y', or 'z'
/[1234567890]/ → any digit
```

Here slashes represent the start and end of a particular expression.

Dash (-)

They are used to specify a range.

For Examples,

```
/[A-Z]/ → matches an uppercase letter
/[a-z]/ → matches a lowercase letter
/[0-9]/ → matches a single digit
```

Caret (^)

They can be used for negation or just to mean ^.

For Examples,

```
/[^a-z]/ → not an lowercase letter
/[^Cc]/ → neither 'C' nor 'c'
/[^.]/ → not a period
/[c^]/ → either 'c' or '^'
```

Question mark (?)

It marks the optionality of the previous expression.

For Examples,

```
/maths?/ → math or maths
/colou?r/ → color or colour
```

Period (.)

Used to specify any character between two expressions.

For Examples,

```
/beg.n/ → Match any character between which fits between beg and n such as begin, begun
```

What are Anchors?

These are special characters that help us to perform string operations either at the beginning or at the end of text input. They are used to assert something about the string or the matching process. Generally, they are not used in a specific word or character but used while we are dealing with more general queries.

Caret character '^'

It specifies the start of the string. For a string to match the pattern, the character followed by the '^' in the pattern should be the first character of the string.

Dollar character '$'

It specifies the end of the string. For a string to match the pattern, the character that precedes the '$' in the pattern should be the last character in the string.

For Examples,

```
^The: Matches a string that starts with 'The'
end$: Matches a string that ends with 'end'
^The end$': Exact string match (starts and ends with 'The end')
roar: Matches a string that has the text roar in it.
```

What are Quantifiers?

Some common Quantifiers are: (*, +, ? and { })

They allow us to mention and control over how many times a specific character(s) pattern should occur in the given text.

Character	Regular-expression meaning
.	Any character, including whitespace or numeric
?	Zero or one of the preceding character
*	Zero or more of the preceding character
+	One or more of the preceding character
^	Negation or complement

Image Source: Google Images

For Example, Let's in a given data, we have variations of the word 'Program' such as 'Program', 'Programmed', 'Programmer'. Our goal is to find only those words where one or more occurrences of 'e' appearing in the word 'Program'. So, to achieve our goal, we can use any of the following quantifiers to indicate the presence of a particular character.

Let's discuss all the quantifiers one by one:

'?'

This quantifier matches the preceding character either zero or one time. Generally, It is used to mark the optional presence of a character.

'*'

This quantifier is used to mark the presence of the preceding character either zero or more times.

'+'

This quantifier matches the preceding character either one or more times. That means for the pattern to match the string, the preceding character has to be present at least once.

{m,n} quantifier

There are four variants of {m, n} quantifier. Let's discuss each of them:

{m, n}

This quantifier matches the preceding character from 'm' number of times to 'n' number of times.

{m, }

{, n}

This quantifier matches the preceding character from zero to 'n' number of times, therefore, the upper limit is fixed for the occurrence of the preceding character.

{n}

This quantifier matches if the preceding character occurs exactly 'n' number of times.

Each of the earlier mentioned quantifiers can be written in the form of {m,n} quantifier in the following way:

- '?' is equivalent to zero or once, or {0, 1}
- '*' is equivalent to zero or more times, or {0, }
- '+' is equivalent to one or more times, or {1, }

For Examples,

```
abc*: matches a string that has 'ab' followed by zero or more 'c'.
abc+: matches 'ab' followed by one or more 'c'
abc?: matches 'ab' followed by zero or one 'c'
abc{2}: matches 'ab' followed by 2 'c'
abc{2, }: matches 'ab' followed by 2 or more 'c'
abc{2, 5}: matches 'ab' followed by 2 upto 5 'c'
a(bc)*: matches 'a' followed by zero or more copies of the sequence 'bc'
```

Whitespace

A white space can include a single space, multiple spaces, tab space, or a newline character (also known as a vertical space). It will match the corresponding spaces in the string.

For Examples,

```
' +', i.e. a space followed by a plus sign will match one or more spaces.
'Chirag Goyal' will allow you to look for the name 'Chirag Goyal' in any given string.
```

Pipe Operator

It's denoted by '|'. It is used as an OR operator. We have to use it inside the parentheses.

For Example, Consider the pattern **'(play|walk)ed' –**

The above pattern will match both the strings — 'played' and 'walked'.

Therefore, the pipe operator tells us about the place inside the parentheses that can be either of the strings or the characters.

Wildcard Character

The '.' (dot) character is also known as the wildcard character. It acts as a placeholder and can match any of the characters.

For Example, if a string starts with three characters, followed by two 0s and three 1s, followed by any four characters. Then, the valid strings that follow the above condition can be

```
cgh00111opqk, kgh00111pksr, etc.
```

Escaping Special Characters

The characters which we discussed above in quantifiers such as '?', '*', '+', '(', ')', '{', etc. can also appear in the input text. So, In such cases to extract these specific characters, we have to use the escape sequences. The escape sequence, represented by a backslash '', is used to escape the special meaning of the special characters.

*****

It is used to escape the star

\.

It is used to escape the dot

\+

It is used to escape the plus sign (to match a '+' sign)

\?

It is used to escape the question mark(to match a question mark)

The '' itself is a special character, and to match the '' character, we have to use the pattern '\\' to escape the backslash.

Character Sets

1. Character sets provide a lot more flexibility than just typing a wildcard or the literal characters. These are groups of characters specified inside square brackets.

2. Character sets can be specified with or without the help of a quantifier.

3. When no quantifier succeeds the character set, it matches only one character and the match is successful only if the character in the string is one of the characters present inside the character set.

For Examples,

Example-1: The pattern **'[a-z]ed'** will match strings such as 'ted', 'bed', 'red' and so on because the first character of each string — 't', 'b' and 'r' — is present inside the range of the character set.

Example-2: When we use a character set with a quantifier, such as **'[a-z]+ed'**, it will match only those words that end with 'ed' like 'watched', 'baked', 'jammed', 'educated', and so on.

From the above examples, we have observed that when a quantifier is present inside the character set, then it loses its special meaning and treated as a normal character.

Let's discuss the two use-cases of '^' :

- **Outside a character set:** works as an anchor
- **Inside a character set:** works as a complement operator, i.e. matches any character other than the ones mentioned inside the character set.

For Example, The pattern **[0–9]** matches any single-digit number but on the contrary, the pattern **'[^ 0-9]'** matches any single digit character that is not a digit.

Meta Sequences

Special Sequences	
\b	Word boundary (zero width)
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])

Image Source: Google Images

We can use meta-sequences in the following two different ways:

Use them without the square brackets.

For Example, the pattern 'w+' will match any alphanumeric character.

Use them inside the square brackets.

For Example, the pattern '[w]+' is the same as 'w+'.

This ends our Part-13 of the Blog Series on Natural Language Processing!

Other Blog Posts by Me

You can also check my previous blog posts.

Previous Data Science Blog posts.

LinkedIn

Here is my Linkedin profile in case you want to connect with me. I'll be happy to be connected with you.

Email

For any queries, you can mail me on Gmail .

End Notes

Thanks for reading!

I hope that you have enjoyed the article. If you like it, share it with your friends also. Something not mentioned or want to share your thoughts? Feel free to comment below And I'll get back to you. 😉

The media shown in this article are not owned by Analytics Vidhya and are used at the Author's discretion.

Part 3: Step by Step Guide to NLP - Text Cleaning and Preprocessing

Part 2: Step by Step Guide to NLP - Knowledge Required to Learn NLP

Part 1: Step by Step Guide to Master NLP - Introduction

blogathon RegEx Regular Expression