Python Regular Expressions Tutorial and Examples: A Simplified Guide

by Selva Prabhakaran (https://www.machinelearningplus.com/author/selva86/) | Posted on January 20, 2018 (https://www.machinelearningplus.com/python/python-regex-tutorial-examples/)

Regular expressions, also called regex, is a syntax or rather a language to search, extract and manipulate specific string patterns from a larger text. It is widely used in projects that involve text validation, NLP and text mining

Regular Expressions in Python: A Simplified Tutorial. Photo by Sarah Crutchfield.

[container] [columnize] 1. Introduction to regular expressions

Contents

2. What is a regex pattern and how to compile one?
3. How to split a string separated by a regex?
4. Finding pattern matches using findall, search and match
4.1 What does regex.findall() do?
4.2 regex.search() vs regex.match()
5. How to substitute one text with another using regex?
6. Regex groups
7. What is greedy matching in regex?
8. Most common regular expression syntax and patterns
9. Regular Expressions Examples
10. Practice Exercises
11. Conclusion

[/columnize] [/container]

1. Introduction to regular expressions

Regular expressions, also called regex is implemented in pretty much every computer language. In python, it is implemented in the standard module `re`.

It is widely used in natural language processing, web applications that require validating string input (like email address) and pretty much most data science projects that involve text mining.

This post is structured into 2 parts.

Before getting to the regular expressions syntax, it's better for you to first understand how the `re` module works.

So, you will first get introduced to the 5 main features of the `re` module and then see how to create commonly used regular expressions in python.

You will see how to construct pretty much any string pattern you will likely need when working on text mining related projects.

[gap]

2. What is a regex pattern and how to compile one?

A regex pattern is a special language used to represent generic text, numbers or symbols so it can be used to extract texts that conform to that pattern.

A basic example is `'\s+'`.

Here the `'\s'` matches any whitespace character. By adding a `'+'` notation at the end will make the pattern match at least 1 or more spaces. So, this pattern will match even tab `'\t'` characters as well.

A larger list of regex patterns comes at the end of this post. But before getting to that, let's see how to compile and play with regular expressions.

```
import re
regex = re.compile('\s+')
```

The above code imports the `'re'` package and compiles a regular expression pattern that can match at least one or more space characters.

[gap]

3. How to split a string separated by a regex?

Let's consider the following piece of text.

```
text = """101 COM    Computers
205 MAT   Mathematics
189 ENG   English"""
```

I have three course items in the format of "[Course Number] [Course Code] [Course Name]". The spacing between the words are not equal.

I want to split these three course items into individual units of numbers and words. How to do that?

This can be split in two ways:

Matplotlib Histogram – How to Visualize Distributions in Python (https://www.machinelearningplus.com/plots/matplot histogram-python-examples/)

ARIMA Model – Complete Guide to Time Series Forecasting in Python (https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/)

Time Series Analysis in Python – A Comprehensive Guide with Examples (https://www.machinelearningplus.com/time-series/time-series-analysis-python/)

Matplotlib Tutorial – A Complete Guide to Python Plot w/ Examples (https://www.machinelearningplus.com/plots/matplot tutorial-complete-guide-python-plot-examples/)

Topic modeling visualization – How to present the results of LDA models? (https://www.machinelearningplus.com/nlp/topic-modeling-visualization-how-to-present-results-lda-models/)

Top 50 matplotlib Visualizations – The Master Plots (with full python code) (https://www.machinelearningplus.com/plots/top-50-matplotlib-visualizations-the-master-plots-python/)

List Comprehensions in Python – My Simplified Guide (https://www.machinelearningplus.com/python/list-comprehensions-in-python/)

Python @Property Explained – How to Use and When? (Full Examples) (https://www.machinelearningplus.com/python/pytho property/)

How Naive Bayes Algorithm Works? (with example and full code) (https://www.machinelearningplus.com/predictive-modeling/how-naive-bayes-algorithm-works-with-example-and-full-code/)

Tags

Bigrams (https://www.machinelearningplus.com/tag/bigrams/) Classification (https://www.machinelearningplus.com/tag/cla Corpus (https://www.machinelearningplus.com/tag/corpus/) Cosine Similarity (https://www.machinelearningplus.com/tag/cosine-similarity/) data.table (https://www.machinelearningplus.com/tag/data-table/) Data Manipulation (https://www.machinelearningplus.com/tag manipulation/) Debugging (https://www.machinelearningplus.com/tag/debugging/) Doc2Vec (https://www.machinelearningplus.com/tag/doc2vec/)

```
<li>By using the re.split method.</li>

<li>By calling the split method of the regex object.</li>
```

```
# split the text around 1 or more space characters
re.split('\s+', text)
# or
regex.split(text)
#> ['101', 'COM', 'Computers', '205', 'MAT', 'Mathematics', '189', 'ENG', 'English']
```

So both these methods work. But which one to use in practice?

If you intend to use a particular pattern multiple times, then you are better off compiling a regular expression rather than using `re.split` over and over again.

[gap]

4. Finding pattern matches using findall, search and match

Let's suppose you want to extract all the course numbers, that is, the numbers 101, 205 and 189 alone from the above text. How to do that?

4.1 What does re.findall() do?

```
# find all numbers within the text
print(text)
regex_num = re.compile('\d+')
regex_num.findall(text)
#> 101 COM    Computers
#> 205 MAT   Mathematics
#> 189 ENG   English
#> ['101', '205', '189']
```

In above code, the special character `'\d'` is a regular expression which matches any digit. I will be covering more such patterns in later in this tutorial.

Adding a `'+'` symbol to it mandates the presence of at least 1 digit to be present in order to be found.

Similar to `'+'`, there is a `'*'` symbol which requires 0 or more digits in order to be found. It practically makes the presence of a digit optional in order to make a match. More on this later.

Finally, the findall method extracts all occurrences of the 1 or more digits from the text and returns them in a list.

4.2 re.search() vs re.match()

As the name suggests, `regex.search()` searches for the pattern in a given text.

But unlike findall which returns the matched portions of the text as a list, `regex.search()` returns a particular match object that contains the starting and ending positions of the first occurrence of the pattern.

Likewise, `regex.match()` also returns a match object. But the difference is, it requires the pattern to be present at the beginning of the text itself.

```
# define the text
text2 = """COM    Computers
205 MAT   Mathematics 189"""

# compile the regex and search the pattern
regex_num = re.compile('\d+')
s = regex_num.search(text2)

print('Starting Position: ', s.start())
print('Ending Position: ', s.end())
print(text2[s.start():s.end()])
#> Starting Position:  17
#> Ending Position:  20
#> 205
```

Alternately, you can get the same output using the group() method of the match object.

```
print(s.group())
#> 205
```

```
m = regex_num.match(text2)
print(m)
#> None
```

[gap]

5. How to substitute one text with another using regex?

To replace texts, use the `regex.sub()` .

Let's consider the following modified version of the courses text. Here I have added an extra tab after each course code.

```
# define the text
text = """101   COM \t  Computers
205   MAT \t  Mathematics
189   ENG  \t  English"""
print(text)
#> 101   COM    Computers
#> 205   MAT     Mathematics
#> 189   ENG     English
```

From the above `text` , I want to even out all the extra spaces and put all the words in one single line.

To do this, you just have to use `regex.sub` to replace the `'\s+'` pattern with a single space `' '`.

```
# replace one or more spaces with single space
regex = re.compile('\s+')
print(regex.sub(' ', text))
# or
print(re.sub('\s+', ' ', text))
#> 101 COM Computers 205 MAT Mathematics 189 ENG English
```

Suppose you only want to get rid of the extra spaces but want to keep the course entries in the new line itself. To achieve that you should use a regex that effectively excludes new line characters but includes all other whitespaces.

This can be done using a negative lookahead `(?!\n)`. It checks for an upcoming newline character and excludes it from the pattern.

```
# get rid of all extra spaces except newline
regex = re.compile('((?!\n)\s+)')
print(regex.sub(' ', text))
#> 101 COM Computers
#> 205 MAT Mathematics
#> 189 ENG English
```

[gap]

6. Regex groups

Regular expression groups is a very useful feature that lets you extract the desired match objects as individual items.

Suppose I want to extract the course number, code and the name as separate items. Without groups, I will have to write something like this.

```
text = """101   COM   Computers
205   MAT   Mathematics
189   ENG    English"""

# 1. extract all course numbers
re.findall('[0-9]+', text)

# 2. extract all course codes
re.findall('[A-Z]{3}', text)

# 3. extract all course names
re.findall('[A-Za-z]{4,}', text)

#> ['101', '205', '189']
#> ['COM', 'MAT', 'ENG']
#> ['Computers', 'Mathematics', 'English']
```

Well, let's see what just happened.

I compiled 3 separate regular expressions one each for matching the course number, code and the name.

For course number, the pattern `[0-9]+` instructs to match all number from 0 to 9. Adding a + symbol at the end makes it look for at least 1 occurrence of numbers 0-9. If you know the course number will certainly have exactly 3 digits, the pattern could have been `[0-9]{3}` instead.

For course code, you can guess that `'[A-Z]{3}'` will match exactly 3 consequtive occurrences of alphabets capital A-Z.

For course name, `'[A-Za-z]{4,}'` will look for upper and lower case alphabets a-z, assuming all course names will have at least 4 or more characters.

Can you guess what would be the pattern if the maximum limit of characters in course name is say, 20?

Now I had to write 3 separate lines to get the individual items. But there is a better way. Regex Groups.

Since all the entries have the same pattern, you can construct a unified pattern for the entire course entry and put the portions you want to extract inside a pair of brackets ().

```
# define the course text pattern groups and extract
course_pattern = '([0-9]+)\s*([A-Z]{3})\s*([A-Za-z]{4,})'
re.findall(course_pattern, text)
#> [('101', 'COM', 'Computers'), ('205', 'MAT', 'Mathematics'), ('189', 'ENG', 'English')]
```

Notice the patterns for the course num: `[0-9]+`, code: `[A-Z]{3}` and name: `[A-Za-z]{4,}` are all placed inside parenthesis `()` in order to form the groups.

[gap]

7. What is greedy matching in regex?

The default behavior of regular expressions is to be greedy. That means it tries to extract as much as possible until it conforms to a pattern even when a smaller part would have been syntactically sufficient.

Let's see an example of a piece of HTML, where I want to retrieve the HTML tag.

```
text = "< body>Regex Greedy Matching Example < /body>"
re.findall('<.*>', text)
#> ['< body>Regex Greedy Matching Example < /body>']
```

Instead of matching till the first occurrence of '>', which I was hoping would happen at the end of first body tag itself, it extracted the whole string. This is the default greedy or 'take it all' behavior of regex.

Lazy matching, on the other hand, 'takes as little as possible'. This can be effected by adding a `?` at the end of the pattern.

```
re.findall('<.*?>', text)
 #> ['< body>', '< /body>']
```

If you want only the first match to be retrieved, use the `search` method instead.

```
re.search('<.*?>', text).group()
 #> '< body>'
```

[gap size="50px"]

8. Most common regular expression syntax and patterns

Now that you understand the how to use the `re` module. Let's see some commonly used wildcard patterns.

[content_band bg_color="#eef" border="all"][container] [custom_headline style="margin: 0;" type="center" level="h2" looks_like="h2" accent="true"]Regular Expressions Syntax[/custom_headline]

```
[custom_headline style="margin: 0;" type="left" level="h6" looks_like="h4" accent="true"]Basic S
.             One character except new line
\.            A period. \ escapes a special character.
\d            One digit
\D            One non-digit
\w            One word character including digits
\W            One non-word character
\s            One whitespace
\S            One non-whitespace
\b            Word boundary
\n            Newline
\t            Tab

[custom_headline style="margin: 0;" type="left" level="h6" looks_like="h4" accent="true"]Modifi
$             End of string
^             Start of string
ab|cd         Matches ab or de.
[ab-d]        One character of: a, b, c, d
[^ab-d]       One character except: a, b, c, d
()            Items within parenthesis are retrieved
(a(bc))       Items within the sub-parenthesis are retrieved

[custom_headline style="margin: 0;" type="left" level="h6" looks_like="h4" accent="true"]Repeti
[ab]{2}       Exactly 2 continuous occurrences of a or b
[ab]{2,5}     2 to 5 continuous occurrences of a or b
[ab]{2,}      2 or more continuous occurrences of a or b
+             One or more
*             Zero or more
?             0 or 1
```

[/container] [/content_band] [gap]

9. Regular Expressions Examples

9.1. Any character except for a new line

```
text = 'machinelearningplus.com (http://machinelearningplus.com)'
print(re.findall('.', text))   # .   Any character except for a new line
print(re.findall('...', text))
 #> ['m', 'a', 'c', 'h', 'i', 'n', 'e', 'l', 'e', 'a', 'r', 'n', 'i', 'n', 'g', 'p', 'l', 'u',
#> ['mac', 'hin', 'ele', 'arn', 'ing', 'plu', 's.c']
```

9.2. A period

```
text = 'machinelearningplus.com (http://machinelearningplus.com)'
print(re.findall('\.', text))   # matches a period
print(re.findall('[^\.]', text))   # matches anything but a period
 #> ['.']
#> ['m', 'a', 'c', 'h', 'i', 'n', 'e', 'l', 'e', 'a', 'r', 'n', 'i', 'n', 'g', 'p', 'l', 'u', '
```

9.3. Any digit

```
text = '01, Jan 2015'
print(re.findall('\d+', text))   # \d  Any digit. The + mandates at least 1 digit.
 #> ['01', '2015']
```

9.4. Anything but a digit

```
text = '01, Jan 2015'
print(re.findall('\D+', text))   # \D  Anything but a digit
 #> [', Jan ']
```

9.5. Any character, including digits

```
text = '01, Jan 2015'
print(re.findall('\w+', text))   # \w  Any character
 #> ['01', 'Jan', '2015']
```

9.6. Anything but a character

```
text = '01, Jan 2015'
print(re.findall('\W+', text))   # \W  Anything but a character
 #> [', ', ' ']
```

9.7. Collection of characters

```
text = '01, Jan 2015'
print(re.findall('[a-zA-Z]+', text))   # [] Matches any character inside
 #> ['Jan']
```

9.8. Match something upto 'n' times

```
text = '01, Jan 2015'
print(re.findall('\d{4}', text))  # {n} Matches repeat n times.
print(re.findall('\d{2,4}', text))
 #> ['2015']
#> ['01', '2015']
```

9.9. Match 1 or more occurrences

```
print(re.findall(r'Co+l', 'So Cooool'))  # Match for 1 or more occurrences
 #> ['Cooool']
```

9.10. Match any number of occurrences (0 or more times)

```
print(re.findall(r'Pi*lani', 'Pilani'))
 #> ['Pilani']
```

9.11. Match exactly zero or one occurrence

```
print(re.findall(r'colou?r', 'color'))
 ['color']
```

9.12. Match word boundaries

Word boundaries `\b` are commonly used to detect and match the beginning or end of a word. That is, one side is a word character and the other side is whitespace and vice versa.

For example, the regex `\btoy` will match the 'toy' in 'toy cat' and not in 'tolstoy'. In order to match the 'toy' in 'tolstoy', you should use `toy\b`

Can you come up with a regex that will match only the first 'toy' in 'play toy broke toys'? (hint: `\b` on both sides)

Likewise, `\B` will match any non-boundary.

For example, `\Btoy\B` will match 'toy' surrounded by words on both sides, as in, 'antoynet'.

```
re.findall(r'\btoy\b', 'play toy broke toys')  # match toy with boundary on both sides
 #> ['toy']
```

[gap]

10. Practice Exercises

Let's get some practice. It's time to open up your python console (https://www.anaconda.com/download/).

[tab_nav type="two-up" float="left"][tab_nav_item title="Question 1" active="true"][tab_nav_item title="Answer" active=""][/tab_nav][tabs][tab active="true"] 1. Extract the user id, domain name and suffix

from the following email addresses.

```
emails = """zuck26@facebook.com
page33@google.com
jeff42@amazon.com"""

desired_output = [('zuck26', 'facebook', 'com'),
 ('page33', 'google', 'com'),
 ('jeff42', 'amazon', 'com')]
```

[/tab][tab]

```
pattern = r'(\w+)@([A-Z0-9]+)\.([A-Z]{2,4})'
re.findall(pattern, emails, flags=re.IGNORECASE)
 #> [('zuck26', 'facebook', 'com'),
 ('page33', 'google', 'com'),
 ('jeff42', 'amazon', 'com')]
```

Use groups with (). There are more sophisticated patterns for matching the email domain and suffix (https://stackoverflow.com/questions/201323/using-a-regular-expression-to-validate-an-email-address?noredirect=1&lq=1). This is just one version of the answer. [/tab][/tabs]

```
 # Solution
pattern = r'(\w+)@([A-Z0-9]+)\.([A-Z]{2,4})'
re.findall(pattern, emails, flags=re.IGNORECASE)
 #>  [('zuck26', 'facebook', 'com'),
#>  ('page33', 'google', 'com'),
#>  ('jeff42', 'amazon', 'com')]

# There are more sophisticated patterns for matching the email domain and suffix (https
```

[gap size="50px"] [tab_nav type="two-up" float="left"][tab_nav_item title="Question 2" active="true"] [tab_nav_item title="Answer" active=""][/tab_nav][tabs][tab active="true"] 2. Retrieve all the words starting with 'b' or 'B' from the following text.

```
text = """Betty bought a bit of butter, But the butter was so bitter, So she bought some better
```

[/tab][tab]

```
import re
re.findall(r'\bB\w+', text, flags=re.IGNORECASE)
 #> ['Betty', 'bought', 'bit', 'butter', 'But', 'butter', 'bitter', 'bought', 'better', 'butter
```

'\b' mandates the left of 'B' is a word boundary, effectively requiring the word to start with 'B'.
Setting 'flags' arg to 're.IGNORECASE' makes the pattern case insensitive.[/tab] [/tabs]

```
 # Solution:
import re
re.findall(r'\bB\w+', text, flags=re.IGNORECASE)
 #> ['Betty', 'bought', 'bit', 'butter', 'But', 'butter', 'bitter', 'bought', 'better', 'butter

# '\b' mandates the left of 'B' is a word boundary, effectively requiring the word to start wit
# Setting 'flags' arg to 're.IGNORECASE' makes the pattern case insensitive.
```

[gap size=”50px”] [tab_nav type=”two-up” float=”left”][tab_nav_item title=”Question 3" active=”true”] [tab_nav_item title=”Answer” active=””][/tab_nav][tabs][tab active=”true”] 3. Split the following irregular sentence into words

```
sentence = """A, very   very; irregular_sentence"""
desired_output = "A very very irregular sentence"
```

[/tab][tab]

```
import re
" ".join(re.split('[;,\s_]+', sentence))
 'A very very irregular sentence'
```

Add more delimiters into the pattern as needed.[/tab][/tabs]

```
 # Solution
import re
" ".join(re.split('[;,\s_]+', sentence))
 #> 'A very very irregular sentence'

 # Add more delimiters into the pattern as needed.
```

[gap size=”50px”] [tab_nav type=”two-up” float=”left”][tab_nav_item title=”Question 4" active=”true”] [tab_nav_item title=”Answer” active=””][/tab_nav][tabs][tab active=”true”] 4. Clean up the following tweet so that it contains only the user’s message. That is, remove all URLs, hashtags, mentions, punctuations, RTs and CCs.

```
tweet = '''Good advice! RT @TheNextWeb: What I would do differently if I was learning to code t

desired_output = 'Good advice What I would do differently if I was learning to code today'
```

[/tab][tab]

```
import re

def clean_tweet(tweet):
    tweet = re.sub('http\S+\s*', '', tweet)  # remove URLs
    tweet = re.sub('RT|cc', '', tweet)  # remove RT and cc
    tweet = re.sub('#\S+', '', tweet)  # remove hashtags
    tweet = re.sub('@\S+', '', tweet)  # remove mentions
    tweet = re.sub('[%s]' % re.escape("""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""), '', tweet)  # re
    tweet = re.sub('\s+', ' ', tweet)  # remove extra whitespace
    return tweet

print(clean_tweet(tweet))
#> Good advice What I would do differently if I was learning to code today
```

[/tab][/tabs]

```
 # Solution
import re
def clean_tweet(tweet):
    tweet = re.sub('http\S+\s*', '', tweet)  # remove URLs
    tweet = re.sub('RT|cc', '', tweet)  # remove RT and cc
    tweet = re.sub('#\S+', '', tweet)  # remove hashtags
    tweet = re.sub('@\S+', '', tweet)  # remove mentions
    tweet = re.sub('[%s]' % re.escape("""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""), '', tweet)  # re
    tweet = re.sub('\s+', ' ', tweet)  # remove extra whitespace
    return tweet

print(clean_tweet(tweet))
 #> Good advice What I would do differently if I was learning to code today
```

[gap size=”50px”] [tab_nav type=”two-up” float=”left”][tab_nav_item title=”Question 5" active=”true”] [tab_nav_item title=”Answer” active=””][/tab_nav][tabs][tab active=”true”] 5. Extract all the text portions between the tags from the following HTML page: https://raw.githubusercontent.com/selva86/datasets/master/sample.html (https://raw.githubusercontent.com/selva86/datasets/master/sample.html)

Code to retrieve the HTML page:

```
import requests
r = requests.get("https://raw.githubusercontent.com/selva86/datasets/master/sample
r.text   # html text is contained here

desired_output = ['Your Title Here', 'Link Name', 'This is a Header', 'This is a Medium Header'
```

[/tab][tab]

```
re.findall('<.*?>(.*)< /.*?>', r.text)  # remove the space after < and /.*> for the pattern to w

 #> ['Your Title Here', 'Link Name', 'This is a Header', 'This is a Medium Header', 'This is a M
```

[/tab][/tabs]

```
# Solution:
# Note: remove the space after < and /.*> for the pattern to work
re.findall('<.*?>(.*)< /.*?>', r.text)
#> ['Your Title Here', 'Link Name', 'This is a Header', 'This is a Medium Header', 'This is a
```

[gap size="50px"]

11. Conclusion

I hope you enjoyed reading this. The purpose of this post was to get you introduced to regular expressions in a simplified way which you remember. Plus, also something you can use as a future reference.