# A simple intro to Regex with Python

We go through some basic examples of using Regex (regular expressions) with Python and show how this framework can be used for powerful text processing.

Tirthajyoti Sarkar

May 19 · 9 min read ★

Image source: Pixabay (free for commercial use)

## Introduction

Text mining is a hot topic in data science these days. The volume, variety, and complexity of textual data are increasing at an astounding space.

As per this article, the global text analytics market was valued at USD 5.46 billion in 2019 and is expected to reach a value of USD 14.84 billion by 2025.

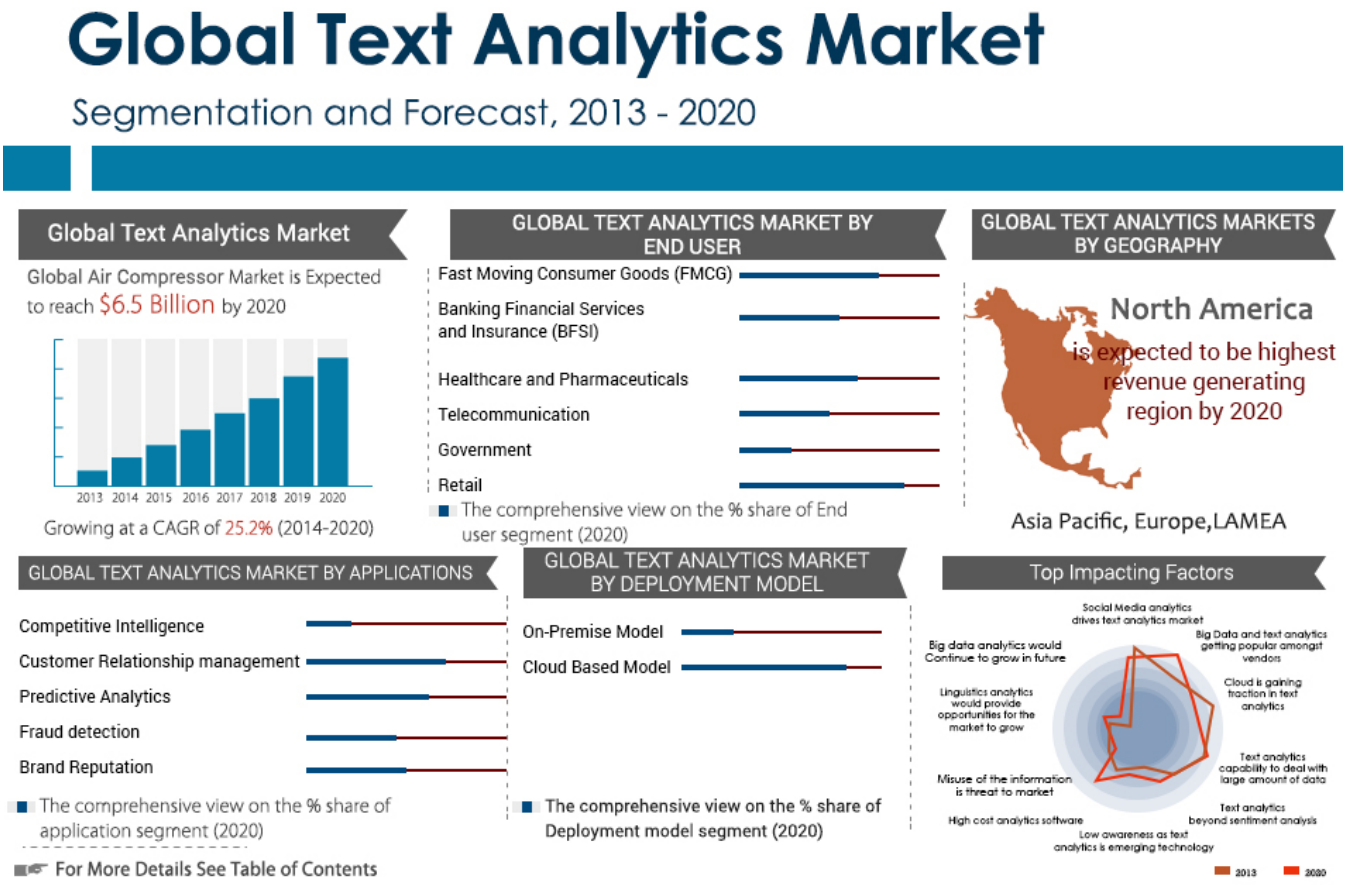


Image source

Regular expressions are used to ***identify*** whether a pattern exists in a given sequence of characters (string) or not and also to ***locate*** the position of the pattern in a corpus of text. They help in manipulating textual data, which is often a pre-requisite for data science projects that involve text analytics.

It is, therefore, important for budding data scientists, to have a preliminary knowledge of this powerful tool, for future projects and analysis tasks.

In Python, there is a built-in module called **re**, which needs to be imported for working with Regex.

```
import re
```

This is the **starting point of the official documentation page**.

In this short review, we will go through the basics of Regex usage in simple text processing with some practical examples in Python.

## The 'match' method

We use the `match` method to check if a pattern matches a string/sequence. It is case-sensitive.

```
string1 = 'Python'
pattern = r"Python"
```

```
if re.match(pattern,string1):
    print("Matches!")
else:
    print("Doesn't match.")
```

```
Matches!
```

```
string2 = 'python'
```

```
if re.match(pattern,string2):
    print("Matches!")
else:
    print("Doesn't match.")
```

```
Doesn't match.
```

## A `compile`d program

Instead of repeating the code, we can use `compile` to create a regex program and use built-in methods.

```
prog = re.compile(pattern)
prog.match(string1)
```

```
<_sre.SRE_Match object; span=(0, 6), match='Python'>
```

So, compiled programs return special object e.g. `match` objects. But if they don't match it will return `None`, and that means we can still run our conditional loop!

```
prog = re.compile(pattern)
if prog.match(string1)!=None:
    print("Matches!")
else:
    print("Doesn't match.")
```

```
Matches!
```

```
if prog.match(string2)!=None:
    print("Matches!")
else:
    print("Doesn't match.")
```

```
Doesn't match.
```

## Positional matching

We can easily use additional parameters in the `match` object to check for positional matching of a string pattern.

```
prog = re.compile(r'y')
```

```
prog.match('Python',pos=1)
```

```
<re.Match object; span=(1, 2), match='y'>
```

```
prog = re.compile(r'thon')
```

```
prog.match('Pythonic',pos=2)
```

```
<re.Match object; span=(2, 6), match='thon'>
```

```
prog.match('Marathon Marathon',pos=4)
```

```
<re.Match object; span=(4, 8), match='thon'>
```

```
prog.match('Marathon Marathon',pos=13)
```

```
<re.Match object; span=(13, 17), match='thon'>
```

Above, we notice that once we created a program `prog` with the pattern `thon`, we can use it any number of times with various strings.

Also, note that the `pos` argument is used to indicate where the matching should be looked into. For the last two code snippets, we change the starting position and get different results in terms of the match. although the string is identical.

## A simple use case

Let's see a use case. We want to find out how many words in a list have the last three letters with 'ing'.

```
prog = re.compile(r'ing')
words = ['Spring','Cycling','Ringtone']
for w in words:
    if prog.match(w,pos=len(w)-3)!=None:
        print("{} has last three letters 'ing'".format(w))
    else:
        print("{} does not have last three letter as 'ing'".format(w))
```

```
Spring has last three letters 'ing'
Cycling has last three letters 'ing'
Ringtone does not have last three letter as 'ing'
```

## The `search` method

For solving the problem above, we could have used a simple string method.

> *What's so powerful about regex?*

The answer is that it can match a very complex pattern. But to see such advanced examples, let's first explore the `search` method.

```
prog = re.compile('ing')
```

```
if prog.match('Spring')==None:
    print("None")
```

```
None
```

```
prog.search('Spring')
```

```
<re.Match object; span=(3, 6), match='ing'>
```

Note, how the `match` method returns `None` (because we did not specify the proper starting position of the pattern in the text) but the `search` method finds the position of the match (by scanning through the text).

Naturally, we can use the `span()` method of the `match` object, returned by `search`, to locate the position of the matched pattern.

```
prog = re.compile(r'ing')
words = ['Spring','Cycling','Ringtone']
for w in words:
    mt = prog.search(w)
    # Span returns a tuple of start and end positions of the match
    start_pos = mt.span()[0] # Starting position of the match
    end_pos = mt.span()[1] # Ending position of the match
    print("The word '{}' contains 'ing' in the position {}-{}".format(w,start_pos,end_pos))
```

```
The word 'Spring' contains 'ing' in the position 3-6
The word 'Cycling' contains 'ing' in the position 4-7
The word 'Ringtone' contains 'ing' in the position 1-4
```

## The `findall` and `finditer` methods

The `search` is powerful but it is also limited to finding the first occurring match in the text. To discover all the matches in a long text, we can use `findall` and `finditer` methods.

The `findall` method returns a list with the matching pattern. You can count the number of items to understand the frequency of the searched term in the text.

The `finditer` method produces an iterator. We can use this to see more information, as shown below.

```
prog.findall('Ringtone of Spring')
```

```
['ing', 'ing']
```

```
lst_of_ring = prog.findall('The phone is singing the ringtone of spring')
print("There are {} occurences of \'ing\' in the text".format(len(lst_of_ring)))
```

```
There are 4 occurences of 'ing' in the text
```

```
for i in prog.finditer('The phone is singing the ringtone of spring'):
    print(i)
```

```
<re.Match object; span=(14, 17), match='ing'>
<re.Match object; span=(17, 20), match='ing'>
<re.Match object; span=(26, 29), match='ing'>
<re.Match object; span=(40, 43), match='ing'>
```

# Wildcard matching (with single characters)

Now, we gently enter the arena where Regex shines through. The most common use of Regex is related to 'wildcard matching' or 'fuzzy matching'. This is where you don't have the full pattern but a portion of it and you still want to find where in a given text, something similar appears.

Here are various examples. Here we will also apply the `group()` method on the object returned by `search` to essentially return the matched string.

## Single-character matching by DOT

Dot `.` matches any single character except the newline character.

```
prog = re.compile(r'py.')
print(prog.search('pygmy').group())
print(prog.search('Jupyter').group())
```

```
pyg
pyt
```

## Lowercase \w to match any single letter, digit or underscore

DOT is limited to alphabetical characters, so we need to expand the repertoire with other tools.

```
prog = re.compile(r'c\wm')
print(prog.search('comedy').group())
print(prog.search('camera').group())
print(prog.search('pac_man').group())
print(prog.search('pac2man').group())
```

```
com
cam
c_m
```

```
c2m
```

## (\W or uppercase W) matches anything not covered with \w

There are symbols other than letter, digits, and underscore. We use \W to catch them.

```
prog = re.compile(r'9\W11')
print(prog.search('9/11 was a terrible day!').group())
print(prog.search('9-11 was a terrible day!').group())
print(prog.search('9.11 was a terrible day!').group())
print(prog.search('Remember the terrible day 09/11?').group())
```

```
9/11
9-11
9.11
9/11
```

## Matching patterns with whitespace characters

\s (lowercase s) matches a single whitespace character like space, newline, tab, return. Naturally, this is used to search for a pattern with whitespace inside it e.g. a pair of words.

```
prog = re.compile(r'Data\swrangling')

print(prog.search("Data wrangling is cool").group())
print("-"*80)
print("Data\twrangling is the full string")
print(prog.search("Data\twrangling is the full string").group())
print("-"*80)

print("Data\nwrangling is the full string")
print(prog.search("Data\nwrangling").group())
```

```
Data wrangling
--------------------------------------------------------------------------------
Data    wrangling is the full string
Data    wrangling
--------------------------------------------------------------------------------
Data
wrangling is the full string
Data
wrangling
```

## \d matches numerical digits 0–9

Here is an example.

```
prog = re.compile(r"score was \d\d")

print(prog.search("My score was 67").group())
print(prog.search("Your score was 73").group())
```

```
score was 67
score was 73
```

And here is an example of a practical application. Suppose, we have a text describing scores of some students in a test. Scores can range from 10–99 i.e. 2 digits. One of the scores is typed wrongly as a 3-digit number (Romie got 72 but it was typed as 721). The following simple code snippet catches it using \d wildcard matching.

```
text = r"""Jack got a 67. I got 73. Romie was close to me - 721.
Sandra scored a whopping 95!"""

digit2 = re.compile(r"\d\d")
digit3 = re.compile(r"\d\d\d")

lines = text.split('.')
```

```
for i,l in enumerate(lines):
    if digit3.search(l) is not None:
        print("There is a typo:", digit3.search(l).group())
    elif digit2.search(l) is not None:
        print("This is a valid score:",digit2.search(l).group())
```

```
This is a valid score: 67
This is a valid score: 73
There is a typo: 721
This is a valid score: 95
```

## Start of a string

The ^(caret) matches pattern at the beginning of a string (but not anywhere else).

```
def print_match(s):
    if prog.search(s)==None:
        print("No match")
```

```
    else:
        print(prog.search(s).group())
```

## ^ (Caret) matches a pattern at the start of the string

```
prog = re.compile(r'^India')

print_match("Russia implemented this law")
print_match("India implemented that law")
print_match("This law was implemented by India")
```

```
No match
India
No match
```

### End of a string

The $ (dollar sign) matches a pattern at the end of the string. Following is a practical example where we are only interested in pulling out the patent information of Apple and discard other companies. We check the end of the text for 'Apple' and only if it matches, we pull out the patent number using the numerical digit matching code we showed earlier.

```
patent_company = re.compile(r'Apple$')
patent_number = re.compile(r'\d\d\d\d\d\d')

s1 = r"Patent no. 123456 belongs to Apple"
s2 = r"Patent no. 345672 belongs to Samsung"
s3 = r"Patent no. 987654 was filed by Apple"
s4 = r"Patent no. 888777 was granted to Microsoft"

for s in [s1,s2,s3,s4]:
    if patent_company.search(s) is not None:
        print("Found a patent of Apple")
        print("Patent number:", patent_number.search(s).group())
    else:
        print("This is not a patent of Apple. Not extracting the number.")
    print("-"*75)
```

```
Found a patent of Apple
Patent number: 123456
---------------------------------------------------------------------------
This is not a patent of Apple. Not extracting the number.
---------------------------------------------------------------------------
Found a patent of Apple
```

```
Found a patent of Apple
Patent number: 987654
--------------------------------------------------------------------------------
This is not a patent of Apple. Not extracting the number.
--------------------------------------------------------------------------------
```

# Wildcard matching (with multiple characters)

Now, we can move on to more complex wildcard matching with multiple characters, which allows us much more power and flexibility.

## Matching 0 or more repetitions

* matches 0 or more repetitions of the preceding regular expression.

```
prog = re.compile(r'ab*')

print_match("a")
print_match("ab")
print_match("abbb")
print_match("b")
print_match("bbab")
print_match("something_abb_something")
```

```
a
ab
abbb
No match
ab
abb
```

## Matching 1 or more repetitions

+ causes the resulting RE to match 1 or more repetitions of the preceding RE.

```
prog = re.compile(r'ab+')

print_match("a")
print_match("ab")
print_match("abbb")
print_match("b")
print_match("bbab")
```

```
print_match("something_abb_something")
```

```
No match
ab
abbb
No match
ab
abb
```

## Matching precisely 0 or 1 repetition

`?` causes the resulting RE to match precisely 0 or 1 repetitions of the preceding RE.

```
prog = re.compile(r'ab?')

print_match("a")
print_match("ab")
print_match("abbb")
print_match("b")
print_match("bbab")
print_match("something_abb_something")
```

```
a
ab
ab
No match
ab
ab
```

## Controlling how many repetitions to match

`{m}` specifies exactly `m` copies of RE to match. Fewer matches cause a non-match and returns `None`.

```
prog = re.compile(r'A{3}')

print_match("ccAAAdd")
print_match("ccAAAAdd")
print_match("ccAAdd")
```

```
AAA
```

```
AAA
No match
```

`{m,n}` specifies exactly `m` to `n` copies of RE to match. Omitting `m` specifies a lower bound of zero, and omitting `n` specifies an infinite upper bound.

```
prog = re.compile(r'A{2,4}B')

print_match("ccAAABdd")
print_match("ccABdd")
print_match("ccAABBBdd")
print_match("ccAAAAAAABdd")
```

```
AAAB
No match
AAB
AAAAB
```

```
prog = re.compile(r'A{,3}B')

print_match("ccAAABdd")
print_match("ccABdd")
print_match("ccAABBBdd")
print_match("ccAAAAAAABdd")
```

```
AAAB
AB
AAB
AAAB
```

```
prog = re.compile(r'A{3,}B')

print_match("ccAAABdd")
print_match("ccABdd")
print_match("ccAABBBdd")
print_match("ccAAAAAAABdd")
```

```
AAAB
No match
No match
AAAAAAAB
```

`{m,n}?` specifies `m` to `n` copies of RE to match in a non-greedy fashion.

```
prog = re.compile(r'A{2,4}')
print_match("AAAAAAA")

prog = re.compile(r'A{2,4}?')
print_match("AAAAAAA")
```

```
AAAA
AA
```

## Sets of matching characters

`[x,y,z]` matches x, y, or z.

```
prog = re.compile(r'[A,B]')
print_match("ccAd")
print_match("ccABd")
print_match("ccXdB")
print_match("ccXdZ")
```

```
A
A
B
```

```
No match
```

## Range of characters inside a set

A range of characters can be matched inside the set. **This is one of the most widely used regex techniques**. We denote range by using a `-`. For example, `a-z` or `A-Z` will match anything between `a` and `z` or `A` and `Z` i.e. the entire English alphabet.

Let's suppose, we want to extract an email id. We put in a pattern matching regex with alphabetical characters + @ + .com. But it cannot catch an email id with some numerical digits in it.

```
prog = re.compile(r'[a-zA-Z]+@+[a-zA-Z]+\.com')

print_match("My email is coolguy@xyz.com")
print_match("My email is coolguy12@xyz.com")
```

```
coolguy@xyz.com
No match
```

So, we expand the regex a little bit. But we are only extracting email ids with the domain name '.com'. So, it cannot catch the emails with other domains.

```
prog = re.compile(r'[a-zA-Z0-9]+@+[a-zA-Z]+\.com')

print_match("My email is coolguy12@xyz.com")
print_match("My email is coolguy12@xyz.org")
```

```
coolguy12@xyz.com
No match
```

It is quite easy to expand on that but clever manipulation of email may prevent extraction of with such a regex.

```
prog = re.compile(r'\w+@+\w+\.+[a-z]{2,4}')
print_match("My email is coolguy12@xy2z.org")
print_match("My email is coolguy12[AT]xyz[DOT]org")
```

```
coolguy12@xy2z.org
No match
```

## Combining the power of Regex by OR-ing

Like any other good computable objects, Regex supports boolean operation to expand its reach and power. OR-ing of individual Regex patterns is particularly interesting.

For example, if we are interested to find phone numbers containing '312' area code, the following code fails to extract it from the second string.

```
prog = re.compile(r'[0-9]{10}')

print_match("3124567897")
print_match("312-456-7897")
```

```
3124567897
No match
```

We can create a combination of Regex objects as follows to expand the power,

```
p0=r'\+*\d*\s[0-9]{3}-[0-9]{3}-[0-9]{4}'
p1= r'[0-9]{10}'
p2=r'[0-9]{3}-[0-9]{3}-[0-9]{4}'
p3 = r'\([0-9]{3}\)[0-9]{3}-[0-9]{4}'
p4 = r'[0-9]{3}\.[0-9]{3}\.[0-9]{4}'
pattern= p0+'|'+p1+'|'+p2+'|'+p3+'|'+p4
prog = re.compile(pattern)

print_match("3124567897")
print_match("312-456-7897")
print_match("(312)456-7897")
print_match("312.456.7897")
print_match("+22 312-456-7897")
```

```
3124567897
312-456-7897
(312)456-7897
312.456.7897
+22 312-456-7897
```

### A combined example

Now, we show an example of extracting valid phone numbers from a text using `findall()` and the multi-character matching tricks we learned so far.

Note that a valid phone number with 312 area code is of the pattern 312-xxx-xxxx or 312.xxx.xxxx.

```
ph_numbers = """Here are some phone numbers.
Pick out the valid phone numbers with 312 area code: \n
312-423-3456, 456-334-6721, 312-5478-9999,
312-Not-a-Number, 777.345.2317, 312.331.6789\n"""

print(ph_numbers)
re.findall('312[-\.][0-9]{3}[-\.][0-9]{4}',ph_numbers)
```

```
Here are some phone numbers.
Pick out the valid phone numbers with 312 area code:

312-423-3456, 456-334-6721, 312-5478-9999,
312-Not-a-Number, 777.345.2317, 312.331.6789


['312-423-3456', '312.331.6789']
```

## The Split method

Finally, we talk about a method that can be used in creative ways to extract meaningful text from an irregular corpus. A simple example is shown below, where we build a Regex pattern with the extrinsic characters which are messing up a regular sentence and use the `split()` method to get rid of those characters from the sentence.

```
sentence = """A, very   very; irregular_sentence"""
print(" ".join(re.split('[;,\s_]+', sentence)))
```

```
A very very irregular sentence
```

## Summary

We reviewed the essentials of defining Regex objects and search patterns with Python and how to use them for extracting patterns from a text corpus.

Regex is a vast topic, with almost being a small programming language in itself. Readers, particularly those who are interested in text analytics, are encouraged to explore this topic more from other authoritative sources. Here are a few links,

**Regular Expressions Demystified: RegEx isn't as hard as it looks**

Are you one of those people who stays away from regular expressions because it looks like a foreign language? I was...

medium.com

For JavaScript enthusiasts,

**An Introduction to Regular Expressions (Regex) In JavaScript**

Regular Expressions aren't an Alien language. Learn the basics of Regex in JavaScript here.

codeburst.io

Top 10 most wanted Regex expressions, ready-made for you,

**Regex cookbook — Most wanted regex**

Top 10 most commonly used (and most wanted) regex

medium.com

. . .

Also, you can check the author's **GitHub repositories** for code, ideas, and resources in machine learning and data science. If you are, like me, passionate about AI/machine learning/data science, please feel free to add me on LinkedIn or follow me on Twitter.

**Tirthajyoti Sarkar - Sr. Principal Engineer - Semiconductor, AI, Machine Learning - ON...**

Making data science/ML concepts easy to understand through writing: https://medium.com/@tirthajyoti Open-source and fun...

www.linkedin.com

Data Science Python Analytics Technology Tech

About Help Legal

Get the Medium app