Introduction to Regular Expression

By Jonathan Feinberg
October 2, 2016

expertanalytics.no

```
"""
Introduction to regular expresion (regex).

The most advanced search utility you never thought you would need.
"""

OVERVIEW = {
    1: "Introduction",
    2: "Editor wars!",
    3: "Simple example",
    4: "Basic regular expresion",
    5: "More advanced regular expresion",
    6: "Real day to day usages",
}
```

```python
"""
The simple substitution example.
"""

source_text = """
All students deserves to pass the course INF3331.
"""

substituted_text = source_text.replace("pass", "fail")

print(substituted_text)
```

```
"""
But it isn't hard to make an example that causes problems.
"""

source_text = """
The target word is apple.

But to make it hard, I will throw in a pineapple.
"""

substituted_text = source_text.replace("apple", "orange")

print(substituted_text)
```

```python
"""
Solution using regular expresion.

(Don't worry if you don't understand the syntax yet.)
"""

import re

source_text = """
The target word is apple.

But to make it hard, I will throw in a pineapple.
"""

substituted_text = re.sub(r"\bapple\b", "orange", source_text)

print(substituted_text)
```

```python
"""
Another example: data extraction.
"""

your_day_of_mill_python_logging_file = """
INFO:  Write code for assignment in INF3331.
DEBUG: Too lazy to document code.
INFO:  Submit code on Github.
ERROR: Assignment fail.
DEBUG: Go home and cry.
"""

for line in your_day_of_mill_python_logging_file.split("\n"):
    if line.startswith("ERROR"):
        print(line)
```

```python
"""
Making life really hard.
"""

more_complex_logging_file = """
ERROR 1.1.1950 10:15 some_module.some_function: Try not to expect the word
ERROR at the beginning of the line.
ERROR 1.1.1950 11:47 some_module.some_function: Log lines can go across
multiple lines."""

for line in more_complex_logging_file.split("\n"):
    if line.startswith("ERROR"):
        print(line)
```

```python
"""
Another solution with regular expression.
Brace yourself, this is going to be messy.
"""

import re

more_complex_logging_file = """
ERROR 1.1.1950 10:15 some_module.some_function: Try not to expect the word
ERROR at the beginning of the line.
ERROR 1.1.1950 11:47 some_module.some_function: Log lines can go across
multiple lines."""

search_string = r"^(ERROR [0-9.]+ [0-9:]+ \w+\.\w+: )"
simplified_logging_file = re.sub(
    search_string, r"@\1", more_complex_logging_file, flags=re.M)

for line in simplified_logging_file.split("@"):
    if line: print(line)
```

```python
"""
Side note: The great editor wars.
"""

EDITOR_OPTIONS = [
    "atom",
    "vim",
    "emacs",
    "pycharm",
]

#%s/\v(e)(m)(a)(c)(s)/\3 \2\1\5\5
```

```python
"""
Let us start from scratch.

Letters and numbers are them self.
"""

import re

spam = "spam"
eggs = "eggs"

search_text = "spammy eggs and eggy spam."

substituted_text = re.sub(spam, eggs, search_text)
print(substituted_text)

search_results = re.findall(spam, search_text)
print(search_results)
```

```python
r"""
The any-key is represented with '.' (except the newline '\n').

The more you know.
"""

import re

search_text = "sing, sang, sung, song, seng."

regex = "s.ng"

search_results = re.findall(regex, search_text)
print(search_results)
```

```
r"""
In between we any-key and literals, we have the character classes and escape
sequences.
--- ----------------- -----------
\w  a word letter     [a-zA-Z0-9]
\W  not a word letter [^a-zA-Z0-9]
\d  a digit           [0-9]
\D  not a digit       [^0-9]
\s  a space           [ \t\n]
\S  not a space       [^ \t\n]
\n  new line
\t  tabular
--- ----------------- -----------
"""
```

```python
"""
Do you have the time?

Note the 'r' prefixes here!
"""

import re

search_text = "The bar is open between 18:04 and 02:00 every friday."
regex = r"\d\d:\d\d"
search_results = re.findall(regex, search_text)
print(search_results)
```

```
"""
Brackets allows for the construction of custom character classes.

------  ----------------------------
Key     Description
------  ----------------------------
[abc]   Range (a or b or c)
[^abc]  Not (a or b or c)
[a-q]   Lower case letter from a to q
[A-Q]   Upper case letter from A to Q
[0-7]   Digit from 0 to 7
------  ----------------------------
"""
```

```python
"""
Partial case insensitive search.
"""

import re

search_text = "Hello, hello, hello, HELLO."

regex = "[hH]ello"

search_results = re.findall(regex, search_text)
print(search_results)
```

```
r"""
Literal versions of special characters has to be cancel.

Special characters:
     ^   [   .   $   {   *   (   \   +   )   /   ?   <   >
Written as:
    \^  \[  \.  \$  \{  \*  \(  \\  \+  \)  \/  \?  \<  \>

Special characters in character class:
     ^   -   ]
Written as:
    \^  \-  \]
"""
```

```python
"""
Lines starts with a '^' anchor.
"""

import re

search_text = "apple 1, apple 2, and apple 3"

regex = "^apple \d"

search_results = re.findall(regex, search_text)

print(search_results)
```

```
r"""
Anchors: All the characters that are not there.

--- ------------------
Key Description
--- ------------------
^   Beginning of line
$   End of line
\b  Boundery of a word
\B  Not a boundery
\<  Left boundery
\>  Right boundery
--- ------------------

Matches before, in between and after characters.
"""
```

```python
"""
Returning to the first example.
"""

import re

source_text = """
The target word is apple.

But to make it hard, I will throw in a pineapple.
"""

substituted_text = re.sub(r"\bapple\b", "orange", source_text)
print(substituted_text)
```

```python
"""
Cavet: When is a newline an actual new line?

Flagging with ‘re.M‘/‘re.MULTILINE‘.
"""

import re

search_text = """apples, oranges, and
pineapples."""

regex = r"^\w+"

search_results = re.findall(regex, search_text)
print(search_results)

search_results = re.findall(regex, search_text, flags=re.M)
print(search_results)
```

```python
"""
The 'maybe'-operator

It is written '?' and indicate that the character is optional.
It is placed behind a character.
"""

import re

search_text = "One egg, many eggs, all the eggses."

regex = "eggs?"

search_results = re.findall(regex, search_text)
print(search_results)
```

```
"""
All the different quantifiers.

------  ----------------------
Quant   Description
------  ----------------------
?       0 or 1
*       0 or more
+       1 or more
{n}     exactly 'n'
{n,}    'n' or more
{,n}    'n' or less
{n,m}   'n', 'm' or in between
------  ----------------------
"""
```

```python
"""
For the words with variable length vowels.
"""

import re

search_text = "no, no, nooooooo."

regex = "no+"

search_results = re.findall(regex, search_text)

print(search_results)
```

```python
"""
Character classes and quantifiers can be used together.
"""

import re

search_text = "One egg, many eggs, all the eggses."

regex = r"egg\w*"

search_results = re.findall(regex, search_text)
print(search_results)
```

```python
"""
How greedy do you want your operators to be?
"""

import re

search_text = "pineappleapplepineapple"

regex = r"\w*apple"
search_results = re.findall(regex, search_text)
print(search_results)
```

```
"""
Non-greedy modifiers.

All quantifiers are greedy; They grab as much as they can.
To make a quantifier non-greedy, add a '?' after it.

------ ----------
Greedy Non-greedy
------ ----------
+      +?
*      *?
?      ??
[a-z]  [a-z]?
------ ----------

Note the difference between the "maybe"-operator '?' and the non-greedy
modifier '?' is the character it follows.
"""
```

```
"""
Once again with and without greedy operator.
"""

import re

search_text = "pineappleapplepineapple"

regex = r"\w*apple"
search_results = re.findall(regex, search_text)
print(search_results)

regex = r"\w*?apple"
search_results = re.findall(regex, search_text)
print(search_results)
```

```python
"""
Group extraction.

For when you are not interested in everything in the regex string.
"""

import re

regex = "(\w*)fix"

search_text = "prefix, infix, postfix, quickfix, fix"

search_results = re.findall(regex, search_text)
print(search_results)
```

```
"""
Multiple extractions at the same time!
"""

import re

regex = r"(\w*)fix(\w*)"

search_text = "prefix, fixpost"

search_results = re.findall(regex, search_text)
print(search_results)
```

```
"""
In substitutions, groups are for extraction.

Captured groups are refered backwards through numbering.
"""

import re

regex_in = r"(\w*)fix"
regex_out = r"\1break"

source_text = "prefix, infix, postfix, quickfix, fix"

substituted_text = re.sub(regex_in, regex_out, source_text)
print(substituted_text)
```

```python
"""
Multiple callbacks in the same substitution.
"""

import re

regex_in = r"(funny)(bunny)"
regex_out = r"\2 \1"

source_text = "Hello funnybunny!"

substituted_text = re.sub(regex_in, regex_out, source_text)
print(substituted_text)
```

```python
"""
Non-capturing groups is encurrage to separete grouping from capturing.
"""

import re

source_text = "mohahahahahahe"

regex_in = r"(\w*?)(?:ha)*(\w*?)"
regex_out = r"\2\1"

substituted_text = re.sub(regex_in, regex_out, source_text)
print(substituted_text)
```

```python
"""
Groups can be used to create multi character alternatives.
"""

import re

search_text = "One egg, many spams, all the hamses."

regex = r"(?:egg|spam|ham)\w*"

search_results = re.findall(regex, search_text)
print(search_results)
```

```python
"""
Bringing grouping, extraction and alternatives together!
"""

import re

regex_in = r"((pre|post)\2?)fix"
regex_out = r"\1break"

source_text = "prefix, preprefix, postpostfix, quickfix, fix"

substituted_text = re.sub(regex_in, regex_out, source_text)
print(substituted_text)
```

```
"""
Returning to a messy example.
"""

import re

more_complex_logging_file = """
ERROR 1.1.1950 10:15 some_module.some_function: Try not to expect the word
ERROR at the beginning of the line.
ERROR 1.1.1950 11:47 some_module.some_function: Log lines can go across
multiple lines."""

search_string = r"^(ERROR [0-9.]+ [0-9:]+ \w+\.\w+: )"
simplified_logging_file = re.sub(
    search_string, r"@\1", more_complex_logging_file, flags=re.M)

for line in simplified_logging_file.split("@"):
    if line: print(line)
```

```
"""
What about the numbers? Can we match those?
"""

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""
```

```python
"""
Naive approach
"""
import re

regex = r"[0-9.eE+\-]+"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Easy first: Integers
"""
import re

regex = r"[\-+]?\d+"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Basic decimals
"""
import re

regex = r"[\-+]?\d+\.\d+"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Basic decimals
"""
import re

regex = r"[\-+]?\d*\.\d*"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Full decimals
"""
import re

regex = r"[\-+]?(\d*\.\d+|\d+\.\d*)"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```
"""
Full decimals
"""
import re

regex = r"[\-+]?(?:\d*\.\d+|\d+\.\d*)"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Integers and decimals all together
"""
import re

regex = r"[\-+]?(?:\d+|\d*\.\d+|\d+\.\d*)"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Integers and decimals in right order.
"""
import re

regex = r"[\-+]?(?:\d*\.\d+|\d+\.\d*|\d+)"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Basic scientific notation.
"""
import re

regex = r"[\-+]?\d+\.\d*[Ee][+\-]\d+"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Expanded scientific notation.
"""
import re

regex = r"[\-+]?\d+\.?\d*[Ee][+\-]\d+"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```python
"""
Full scientific notation.
"""
import re

regex = r"[\-+]?\d+\.?\d*[Ee][+\-]?\d+"

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```

```
"""
Full scientific notation.
"""
import re

sign = r"[\-+]?"
decimal = r"(?:\d*\.\d+|\d+\.\d*|\d+)"
science = r"(?:[Ee][+\-]?\d+)?"
regex = sign + decimal + science

all_the_numbers = """
INTEGERS: 1 +2 -3
DECIMAL: +42.5 -.25 3.
SCIENCE: .23E+4 -4.00e-02 +1e1
CORNER_CASES: 1-2 4.- -E1- C++ .
"""

print(re.findall(regex, all_the_numbers))
```