Programming

with python™

By

Rami Tailakh

Senior Software Engineer and Data Science Practitioner

MSc in Applied Computing and Information Technology

Day-6 Agenda

- Day-5 Quick Review

- Regular Expressions (Regex)

Regular Expressions - REGEX

- Regular expression (re, regex) is a specialized programming language embedded inside Python
- Regex is a sequence of characters that defines a search pattern (or rules) to be specify a set of possible strings required to match
- In other words, this pattern can be used to <u>**find**</u> or <u>**find and replace**</u> on strings
- Regex is made available through the **“re”** module

Regex Applications

- Search Tools
- Text processing: Find and Replace
- Text analysis
- Matching and Validating
- Text Extraction

Regex Characters

- Regular (ordinary) characters match themselves exactly and do not have a special meaning in their regular expression syntax, e.g.:

```
>>> re.search('abc','123abcdef321')
```

- A metacharacter (special) has a special meaning; e.g.:

^ . [ ] ? \w \d + * ()

```
>>> re.search('[\w.-]+@','123ABCD@efg.hi')
```

Regex Metacharacters

Metacharacter	Description
^	Matches the start of the string
.	Matches a single character, except a newline But when used inside square brackets, a dot is matched
[ ]	A bracket expression matches a single character from the ones inside it [abc] matches ‘a’, ‘b’, and ‘c’ [a-z] matches characters from ‘a’ to ‘z’ [a-cx-z] matches ‘a’, ’b’, ’c’, ’x’, ’y’, and ‘z’
[^]	Matches a single character from those except the ones mentioned in the brackets[^abc] matches all characters except ‘a’, ‘b’ and ‘c’

Regex Metacharacters-Continue

Metacharacter	Description
( )	Parentheses define a marked subexpression, also called a block, or a capturing group
\t, \n, \r, \f	Tab, newline, return, form feed
*	Matches the preceding character zero or more times ab*c matches ‘ac’, ‘abc’, ‘abbc’, and so on [ab]* matches ‘’, ‘a’, ‘b’, ‘ab’, ‘ba’, ‘aba’, and so on (ab)* matches ‘’, ‘ab’, ‘abab’, ‘ababab’, and so on
{m,n}	Matches the preceding character minimum m times, and maximum n times a{2,4} matches ‘aa’, ‘aaa’, and ‘aaaa’
{m}	Matches the preceding character exactly m times

Regex Metacharacters-Continue

Metacharacter	Description
?	Matches the preceding character zero or one times ab?c matches ‘ac’ or ‘abc’
+	Matches the preceding character one or one times ab+c matches ‘abc’, ‘abbc’, ‘abbbc’, and so on, but not ‘ac’
|	The choice operator matches either the expression before it, or the one after abc|def matches ‘abc’ or ‘def’
\w	Matches a word character (a-zA-Z0-9) \W matches single non-word characters
\b	Matches the boundary between word and non-word characters

Regex Metacharacters-Continue

Metacharacter	Description
\s	Matches a single whitespace character \S matches a single non-whitespace character
\d	Matches a single decimal digit character (0-9)
\	A single backslash inhibits a character's specialness Examples- \. \\ * When unsure if a character has a special meaning, put a \ before it: \@
$	A dollar matches the end of the string

Regex Functions - match()

- takes two arguments- a pattern and a string
- It matches a pattern to a string
- A string returned when matched, otherwise, None

```
>>> re.match('......\d','python3')
```

Out: **<re.Match object; span=(0, 7), match='python3'>**

```
>>> print(re.match('......\d','python-3'))
```

Out: **None**

Regex Functions - search()

- It also takes a pattern and a string
- A string is searched according to a pattern
- The search stops at the first match

```
>>> print(re.search('^Python', 'I am practicing Python
programming'))
```

Out: **None**

```
>>> print(re.search('^Python', 'Python is easy))
```

Out: **<re.Match object; span=(0, 6), match='Python'>**

Regex Functions - findall()

- It also takes a pattern and a string
- findall() returns a list of all matches found

```
>>> match_list=re.findall('\w*ing','Hello! I am studying Python programming and practicing challenging examples')

>>> match_list
```

Out: **['studying', 'programming', 'practicing', 'challenging']**

Regex Functions - sub()

- It is to substitute the part of a string with another
- The sub() function takes three arguments: pattern, substring, and string.

```
>>> re.sub('[#@.,;:()?!]','',sample_text)
```

Out: **<All special characters are removed, right?>**

Regex Functions - compile()

- It helps to use a pattern again without rewriting it

```
>>> pattern = re.compile('[#@.,;:()?!]')

>>> pattern.sub('',sample_text)
```

Out: **<All special characters are removed, right?>**

Regex Functions - match vs search

- **match**: finds something **at the beginning** of s string and returns a match object
- **search**: finds something **anywhere** in a string and returns a match object.

```
# TRY THE FOLLOWING LINES:

>>> string = "123abc"

>>> re.match("[a-z]+",string)

>>> re.search("[a-z]+",string)
```

Regex Functions-Example 1

Write a regex that extracts an email address from a string.

```
>>>matched_email=re.search(r'[\w.-]+@[\w-]+\.[\w]+','Our contact email is info@newsoft.ps')

>>>matched_email.group()
```

Out: **info@newsoft.ps**

Try with: `matched_email.group(1)`, `matched_email.group(2)`, and `matched_email.group(3)`

Regex Functions-Example 2

Write a regex that extracts an email address from a string.

```
>>>matched_email=re.search(r'[\w.-]+@[\w-]+\.[\w]+','Our contact email is info@newsoft.ps')

>>>matched_email.group()
```

Out: **info@newsoft.ps**

Regex Functions-Example 3

Remove repetition of a character in a string

```
>>> re.sub(r'i+','i','ramiii' )
```

Out: **rami**

Regex Options - IGNORECASE

- This re option is to ignore the case while matching

```
>>> sample_text = 'XML parsing is easy now, however,
parsing json in Python is easier than parsing Xml files'

>>>match_list=re.findall(r'xml',sample_text,re.IGNORECASE)

>>> match_list
```

Out: **['XML', 'Xml']**

Regex Options - MULTILINE

- This allows ^ and $ to match the start and end of each line, when processing a string of multiple lines
- It handles each line instead of the whole string

```
>>> string = """Python
Java
Ruby"""

# TRY THE FOLLOWING LINES
>>> print(re.findall(r"^\w", string))
>>> print(re.findall(r"^\w", string, re.MULTILINE)

>>> print(re.findall(r"\w$", string))
>>> print(re.findall(r"\w$", string, re.MULTILINE))
```

Regex Options - DOTALL

- In a multiline string, the first line is only matched
- DOTALL option is used to work with the whole string even it is a multiline
- Simply, it makes the ‘.’ special character match all characters including newline characters

Regex Options - DOTALL-Continue

```
>>> re.sub(r'i+','i','ramiii' )

text = '''
   <ELEMENTS>
       <ELEMENT>I am Element-1 with multilines.
       This is Line-1.
       This is Line-2.
       </ELEMENT>
       <ELEMENT>I am Element-2 with multilines.
       This is Line-1.
       This is Line-2.
       </ELEMENT>
   </ELEMENTS>
   This is not important

'''

>>> re.search(r'<ELEMENTS>.*<ELEMENT>', text, re.DOTALL)

# What do you observe?
```

Greedy vs Non-Greedy

- The metacharacters: *, +, and **?** are to keep searching in a string
- The **.*** is greedy
- The **?** makes it non-greedy.
- **{m,n}** is to search for matches as few as possible

```
# TRY WITH THE FOLLOWING LINE:

>>> re.findall(r'</?\w+>','<img>Image</img> <i>Italic</i>
<strong>Strong</strong>')
```

Challenge

1. Write a python function that detects floating number from a string
2. Write a python function that validates a Jawwal number.
3. Write a python function that validates a URL.
4. Write a python function that validates an email address.
5. Write a python function that detects dates from a string.
6. From a text file, extract all words with 7 letters and more.