Regular Expressions in Python

Created for

Компютърно образователен център
Progress
Палитра от възможности!

Iva E. Popova on LinkedIn

Regular Expressions Basics

Overview

Common Use Cases for Regex

- Validation
  - Verify if a string matches a certain pattern, like email or phone number validation.
- Data Extraction
  - Extract specific information from a text, such as extracting URLs from a webpage.
- Data Cleaning
  - Remove unwanted characters or format text data according to predefined rules.
- Text Parsing
  - Splitting text into tokens based on specified patterns.

Regex Symbols

literal and meta symbols

- A regex grammar includes 2 types of symbols:
  - Regular symbols: they are matched literally on the matching string
  - Meta (special) characters: they have special meaning and gives the power of regex

complete list of regex metacharacters Copy

```
. ^ $ * + ? { } [ ] \ | ( )
```

- If we want to match **literally** a meta character we have to escape it with backslash `'\'`

Python Copy

```
import re

text = "try to match: 2+3"
rx = re.compile('2\+3')

res = rx.search(text)
if res:
    print( res.group())
```

Character classes and sets

Overview

- The square brackets are used to define a character set. Like: `[abc]` (will match 'a' or 'b' or 'c').
- The character set itself match only one symbol!
- Symbols inside brackets are the elements of set.
- The hyphen (`-`), when it is between 2 symbols, has special meaning inside the character class - it defines a range. Like: `[0-9]`. If it is in the end, it is considered as a hyphen.

Character set	Description
`[abc]`	Match any one of the symbols listed ('a' or 'b' or 'c')
`[a-z]`	Match any symbol, from 'a' till 'z' (i.e. any lower Latin letter)
`[0-9]`	Match any digit
`[0-9-]`	Match any digit or hyphen
`[^abc]`	Match any symbol, except 'a or 'b' or 'c' (i.e. the `^` negates the characters in the set)

Character Sets examples

Python Copy

```
import re

# Match any vowel character
matched = re.findall(r'[aeiouy]','astroid' );
print(matched)
#OUTPUT: ['a', 'o', 'i']

# Match any non-vowel character
matched = re.findall(r'[^aeiouy]','astroid' );
print(matched)
#OUTPUT: ['s', 't', 'r', 'd']

# match any digit or hyphen:
matched = re.findall('[0-9-]', 'a2-b8');
print(matched)
#OUTPUT: ['2', '-', '8']
```

Overview

- Character classes in regular expressions are shorthands for commonly used sets of characters. They provide a concise way to match specific types of characters.
- Some commonly used character classes include:
  `\d`: Matches any digit character (equivalent to `[0-9]`).
  `\w`: Matches any word character (equivalent to `[a-zA-Z0-9_]`).
  `\s`: Matches any whitespace character (spaces, tabs, newlines, etc.).
  `\D`: Matches any non-digit character (equivalent to `[^0-9]`).
  `\W`: Matches any non-word character (equivalent to `[^a-zA-Z0-9_]`).
  `\S`: Matches any non-whitespace character.

Character classes example

Python

Copy

```python
import re

# Match any digit character
text = "The price is $25.99."
pattern = re.compile(r'\d')
result = pattern.findall(text)
print(result)
# Output: ['2', '5', '9', '9']

# Match any word character
text = "The quick brown fox jumps over the lazy dog."
pattern = re.compile(r'\w+')
result = pattern.findall(text)
print(result)
# Output: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

# Match 'line' followed by digit followed by whitespace:
text = """line1
line2
line3 line4"""

pattern = re.compile(r'line\d\s')
result = pattern.findall(text)
print(result)

# Output: ['line1\n', 'line2\n', 'line3 ']
```

Quantifiers

Overview

- Quantifiers in regular expressions specify the quantity of the preceding element to match.

Quantifier	Description
r *	*r* match **0** or **more times**
r +	*r* match **1** or **more times**
r ?	*r* match **0** or **1**time
r {n}	*r* match **exactly *n* times**
r {n,m}	*r* match between **n** and **m** times (n, m are positive)

r can be character, group, or character class/set

Examples

- *
  - Matches zero or more occurrences of the preceding character or group.

```python
pattern = re.compile(r'ab*c')
result = pattern.match('ac')      # Matches
result = pattern.match('abc')     # Matches
result = pattern.match('abbc')    # Matches
result = pattern.match('a')       # Matches
```

- +
  - Matches one or more occurrences of the preceding character or group.

```python
pattern = re.compile(r'ab+c')
result = pattern.match('ac')      # Doesn't match
result = pattern.match('abc')     # Matches
result = pattern.match('abbc')    # Matches
result = pattern.match('a')       # Doesn't match
```

- ?
  - Matches zero or one occurrence of the preceding character or group.

```python
pattern = re.compile(r'ab?c')
result = pattern.match('ac')      # Matches
result = pattern.match('abc')     # Matches
result = pattern.match('abbc')    # Doesn't match
result = pattern.match('a')       # Doesn't match
```

Examples

- {n}
  - Matches exactly n occurrences of the preceding character or group.

```python
pattern = re.compile(r'a{2}b')
result = pattern.match('aab')     # Matches
result = pattern.match('ab')      # Doesn't match
result = pattern.match('aaab')    # Doesn't match
```

- {n,}
  - Matches n or more occurrences of the preceding character or group.

```python
pattern = re.compile(r'a{2,}b')
result = pattern.match('aab')     # Matches
result = pattern.match('aaab')    # Matches
result = pattern.match('ab')      # Doesn't match
```

- {n,m}
  - Matches at least n and at most m occurrences of the preceding character or group.

```python
pattern = re.compile(r'a{2,3}b')
result = pattern.match('aab')     # Matches
result = pattern.match('aaab')    # Matches
result = pattern.match('aaaab')   # Doesn't match
result = pattern.match('ab')      # Doesn't match
```

Quantifiers (greedy and non-greedy match)

Anchors and Boundaries

Overview

- They specify a **position** in the string where a match should occurs.
- They are zero-width, i.e.when matched they do NOT consume characters from the string.

Anchor	Description
^	Matches the beginning of the string (or the line, if **m** flag is used)
$	Matches the end of the string (or the line, if **m** flag is used)
\b	Matches on word boundaries, i.e. between *word*(\w) and *non-word*(\W) characters. Note that the **start** and **end** of string are considered as non-word characters.
\Z	Matches only at the end of the string.

^ and $ example

Python Copy

```
import re

strings = [
    'ana',
    'ana bel',
]
rx = re.compile(r'^a.+a$');

for string in strings:
    res = rx.findall(string)
    print("{} matches in {}".format(len(res), string))

#Output:
#1 matches in ana
#0 matches in ana bel
```

\b example

Python Copy

```
import re

strings = [
    '',
    'a',
    '@',
    '@a',
    'aa',
    'a!',
    'a,a',
]
rx = re.compile(r'\b');

for string in strings:
    res = rx.findall(string)
    print(f"{len(res)} word bounders counted in {string}")


# 0 word bounders counted in
# 2 word bounders counted in a
# 0 word bounders counted in @
# 2 word bounders counted in @a
# 2 word bounders counted in aa
# 2 word bounders counted in a!
# 4 word bounders counted in a,a
```

Modifiers/Flags

Modifiers/Flags

- Flags reflects how the regular expression is executed.
- They are available in the re module with a long name such as `re.IGNORECASE` or with a short, one-letter form such as `re.I`.
- Multiple flags can be specified by bitwise OR-ing them. For example `re.I | re.M` sets both the I and M flags.
- Flags can be set as second argument to the re.compile() method

Python Copy

```
regex_with_flags_arg = re.compile(r"[aeiouy]+",re.I)
```

- Or as a third argument on re module methods

Python Copy

```
m = re.search(r"[aeiouy]+", str, re.I)
```

- Flags can be also set in the regular expression itself, using (?aiLmsux) syntax at the beginning of the regex string

Terminal Copy

```
regex_with_flags_prefix= re.compile(r"(?i)[aeiouy]+")
```

Modifiers/Flags list

In reges	As param	Description
(?i)	re.I	case-**i**nsensitive matching
(?m)	re.M	**m**ultiline matching
(?s)	re.S	Make the '**.**' to match any character at all, including a newline
(?x)	re.X	Allows to write readable regexes by using spaces and comments('#') in the regex. More on: re.X

Reference: Compilation flags

Modifiers/Flags example

- Using the re.IGNORECASE flag to perform case-insensitive matching

Python Copy

```python
import re

text = "The quick brown fox jumps over the lazy dog."

pattern = re.compile(r"the", flags=re.IGNORECASE)
result = pattern.findall(text)


# Output:
# ['The', 'the']
```

- Using re.MULTILINE flag to match at beginning of each line:

Python Copy

```python
import re

# Example text containing multiple lines
text = """Line 1
# Line 2
Line 3"""

# Matching lines starting with "Line"
pattern = re.compile(r'^Line\s*\d', re.MULTILINE)
result = pattern.findall(text)
print(result)

# Output: ['Line 1', 'Line 3']
```

Alternation

Alternation

- Alternation allows you to match one pattern or another. It is denoted by the pipe symbol `|`.

Python Copy

```
import re

text = "I love cats. He love dogs."
pattern = re.compile(r"cat|dog")

result = pattern.findall(text)
print(result)

# Output: ['cat', 'dog']
```

- The alternation operator has the lowest precedence of all regex operators. That is, it tells the regex engine to match either everything to the left of pipe, or everything to the right of pipe.
  - If you want to limit the reach of the alternation, you need to use parentheses for grouping (discussed in next topic).

Python Copy

```
import re

text = "I love cats. He love dogs."
pattern = re.compile(r"(?:cat|dog)s")

result = pattern.findall(text)
print(result)
```

Grouping and capturing

Overview

- Brackets: (and), play a dual role in regex!
- They can be used to capturing groups, to remember a matched part of the string. Like:

Python Copy

```python
import re

text = "Ivan Ivanov: 30 years old, Petar Petrov: 25 years old"

# Using capturing group to get names and ages
pattern = re.compile(r"(\w+ \w+): (\d+) years old")
matches = pattern.findall(text)

for match in matches:
    print("Name:", match[0])
    print("Age:", match[1])
```

- Brackets can be used for non-capturing groups to group parts in a regex without capturing:
  - **`/(?:r1|r2)r3/`** => match **`r1r3`** OR **`r2r3`**, but not **`r1r2r3`**

Python Copy

```python
import re

# Example text
text = "I love strawberries and raspberries, but not blueberries."

# Using non-capturing group with alternation to match "quick" or "lazy"
pattern = re.compile(r"(?:straw|rasp)berries")
result = pattern.findall(text)
print(result)
# Output: ['strawberries', 'raspberries']
```

- NB! Capturing is slow and memory consuming! If you need the parenthesis just for grouping- always use the **?:** prefix.

Using regex in Python - the `re` module

Overview

- The built-in `re` module in Python provides regular expression matching operations similar to those found in Perl.
- Regular expressions are compiled into Regular Expression Object, which have methods for various operations such as searching for pattern matches or performing string substitutions.
- REs in Python are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them.

How to write regex

- Regex in Python are written as string, which can be passed to re.compile() method or directly to other matching methods, like re.search(), re.match()
- We can use any string literals, including the raw string syntax (`r''`)
- Example of raw string:

Python Copy

```
print('\test')      #        est
print(r'\test')     #\test
print('\\test')     #\test
```

the `re.compile()` method

- Compiles a regular expression pattern into a regular expression object, which can be used for matching using its methods for matching and search

Python Copy

```
import re

text = "ABRACADABRA"

regex = re.compile(r'aca', re.I)

if regex.search(text):
    print('Match')
```

Regular Expression Objects Methods

regex.search(string[, start[, end]]))

- Scan through string looking for where this regular expression produces a match.
- If match produced => returns a corresponding match object
- If string does not matches the pattern => return None
- optional parameters:
  - *start* - the index where the search should start
  - *end* - the index where the search should ends
- Example:

Python Copy

```
import re

text = "123abc456"
rx = re.compile('abc')

res = rx.search(text) # will match
res = rx.search(text,3) # will match, 'a' is on index 3 in text
res = rx.search(text,4) # would NOT match
```

regex.match(string[, start[, end]])

- Matches only on the beginning of the string
- If you want to locate a match anywhere in string, use search() instead
- Reference: search-vs-match @python docs
- Example:

Python Copy

```
text = "123abc456"
rx = re.compile('abc')

res = rx.match(text)

res = rx.match(text) # will NOT match, 'abc' is not in the
beginning
res = rx.match(text,3) # will match, as matching starts from index
3
```

regex.findall(string[, pos[, endpos]])

- Returns a list of strings containing all non-overlapping matches of regex in the string
- The string is scanned left-to-right, and matches are returned in the order found
- Example:

Python Copy

```
text = "123abc456abcabc"
rx = re.compile('abc')

res = rx.findall(text) # ['abc', 'abc', 'abc']
res = rx.findall(\dtext) # ['3abc', '6abc']
```

Other Matching Methods

- regex.finditer(string[, pos[, endpos]])
- regex.fullmatch(string[, pos[, endpos]])
- regex.sub(repl, string, count=0)
- regex.subn(repl, string, count=0)

re module-level functions

- The methods described above, was methods of an Regular Expression Objects
- Python has similar functionality defined in the `re` module, like:
  - re.search(pattern, string, flags=0)
  - re.match(pattern, string, flags=0)
  - and so on...
- The difference is that we must pass the pattern string as first argument, and optional flags at the end.

The Match Object

Overview

- match() and search() methods returns a Match Object or None
- So you can test whether there was a match with a simple if statement:

Python Copy

```
match = re.search(pattern, string)
if match:
    process(match)
```

Match Object Methods

Method/Attribute	Purpose
group()	Return the string matched by the RE
groups()	Return a tuple containing all the subgroups of the match
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

More methods: Match Object

Match Object Methods - example

Python

Copy

```python
import re

text = "Name: Maria, Age: 30"
pattern = r"Name: (\w+), Age: (\d+)"  # Capture name and age

match = re.search(pattern, text)

if match:
    print("Full match:", match.group(0))
    print("Name (Group 1):", match.group(1))
    print("Age (Group 2):", match.group(2))
    print("All groups:", match.groups())

# OUTPUT
# Full match: Name: Maria, Age: 30
# Name (Group 1): Maria
# Age (Group 2): 30
# All groups: ('Maria', '30')
```

Examples

Validate Bulgarian mobile number

Python Copy

```python
import re

def is_valid_bg_mobile_number(number):
    """ Validates a Bulgarian mobile number.
        Note:
            The format of a valid Bulgarian mobile number is: +359 XX XDDD DDD,
            where X is in [7, 8, 9], and D is in [0-9].
        Args:
            number (str): The mobile number to be validated.

        Returns:
            bool: True if the number is a valid Bulgarian mobile number, False otherwise.
    """
    rg = re.compile(r'^\+359\s[7-9]{2}\s[7-9]\d{3}\s\d{3}$')

    m = rg.match(number)

    return True if m else False


if __name__=="__main__":
    phone_numbers = [
        '+359 88 7123 456', #yes
        '+359 88 7123456',  #no
        '+359 88 1123 456', #no
        '+359 87 9123 456'  #yes
    ]

    for number in phone_numbers:
        if is_valid_bg_mobile_number(number):
            print(f'{number:18} #yes')
        else:
            print(f'{number:18} #no')
```

Validate user name

Python Copy

```python
import re

def is_valid_user_name(number):
    """ Validates a User name.

        Note:
            User name must follow next rules:
                1. Must consists of 3 to 10 characters inclusive.
                2. Username can only contain alphanumeric characters, dashes (-) and underscores (_).
                3. The first character of the username must be an alphabetic character

        Args:
            number (str): The user name to be validated.

        Returns:
            bool: True if the name is a valid, False otherwise.
    """
    rg = re.compile(r'''
    ^                   # beginning of string
        [a-zA-Z]        # rule 3
        [\w-]{2,9}      # rule 1 and 2
    $                   # end of string
    ''', re.VERBOSE)
    m = rg.match(number)

    return True if m else False


if __name__=="__main__":
    user_names = [
        "ada",          # yes
        "a__",          # yes
        "a12345",       # yes
        "a1234567890",  # no (rule 1)
        "1aaaaaaa",     # no (rule 3)
        "aaa#",         # no (rule 2)
        "a",            # no (rule 1)
    ]
```

Resources

Online Regex Testers and Cheet-sheet

- Python Regex Cheatsheet
- regex101.com - Online regex tester and debugger: PHP, PCRE, Python, Golang ...

regex Excersises

- regex101 regex quiz
- regexone.com - Learn Regular Expressions with simple, interactive exercises.

These slides are based on
customised version of
Hakimel's reveal.js
framework