

04-Python intermediate

Regular expression in Pthon

Posted on September 19, 2021

Last updated on February 14, 2023

ALL Pdf Notes (<https://github.com/amrit94/DataScience/tree/main/notes>)

4. Regular expression

- A Regular Expression (RegEx) is a sequence of characters that defines a search pattern
- RegEx can be used to check if a string contains the specified search pattern.
- Python has a built-in package called `re`, which can be used to work with Regular Expressions.

If we want to represent a group of Strings according to a particular format/pattern then we should go for Regular Expressions.

4.1 Raw string

- Raw string do not treat backslashes(`\`) as a part of sequence
 - It will be printed normally as a result
-

```
print(r"Hello\tfrom AskPython\nHi")
# Hello\tfrom AskPython\nHi

print("Hello\tfrom AskPython\nHi")
# Hello from AskPython
# Hi
```

4.2 String operation

- `in`, `find`, `index`

```
string = 'foo123bar'

print('123' in string)
# True

print(string.find('123'))
# 3

print(string.index('123'))
# 3
```

4.3 RegEx function

Function	Description
<code>re.search()</code>	Scans a string for a regex match
<code>re.match()</code>	Looks for a regex match at the beginning of a string
<code>re.fullmatch()</code>	Looks for a regex match on an entire string
<code>re.findall()</code>	Returns a list of all regex matches in a string
<code>re.finditer()</code>	Returns an iterator that yields regex matches from a string

re.match()

We can use match function to check the given pattern at beginning of target string. If the match is available then we will get Match object, otherwise we will get None.

Match doesn't works for multi-line string

```
# For match --> returns matched object
# No match --> return None

re.match('string', 'mystring')
# None
re.match('xyz', 'string xyz')
# None
re.match('string', 'stringxyz')
# <re.Match object; span=(0, 6), match='string'>
```

re.search()

We can use `search()` function to search the given substring in the target string.

If the match is available then it returns the Match object which represents first occurrence of the match. If the match is not available then it returns `None`.

Also works for multi-line string

```
string = '''We are learning regex
string matching'''
re.search('string', string)
# <re.Match object; span=(22, 28), match='string'>
```

re.compile()

- `re.compile(<regex>, flags=0)`
- Compiles a regex into a regular expression object.

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```

import re

# without compile
print(re.search('ab', "abaababa"))
# <_sre.SRE_Match object; span=(0, 2), match='ab'>

# Two ways to compile

# Way 1
pattern=re.compile("ab")
print(re.search(pattern, "abaababa"))
# <_sre.SRE_Match object; span=(0, 2), match='ab'>

# Way 2
print(pattern.search("abaababa"))
# <_sre.SRE_Match object; span=(0, 2), match='ab'>

```

re.fullmatch()

- re.fullmatch(<regex>, <string>, flags=0)

We can use fullmatch() function to match a pattern to all of target string. i.e complete string should be matched according to given pattern.

If complete string matched then this function returns Match object otherwise it returns None.

```

import re

print(re.fullmatch(r'\d+', '123'))
# <re.Match object; span=(0, 3), match='123'>

print(re.fullmatch(r'\d+', '123foo'))
# None

print(re.match(r'\d+', '123foo'))
# <re.Match object; span=(0, 3), match='123'>

```

re.findall()

- re.findall(<regex>, <string>, flags=0)

- Returns a list of all matches of a regex in a string.
- Return all non-overlapping matches of pattern in string, as a list of strings.
- The string is scanned left-to-right, and matches are returned in the order found.

```
print(re.findall('pk', 'th thpkd is pk dsspk'))
# ['pk', 'pk', 'pk']

print(re.findall(r'(\w+),(\w+)', 'foo,bar,baz,qux,quux,corge'))
# [('foo', 'bar'), ('baz', 'qux'), ('quux', 'corge')]
```

re.finditer()

- re.finditer(<regex>, <string>, flags=0)
- Returns the iterator yielding a match object for each match.
- On each match object we can call start(), end() and group() functions.

```
import re
itr = re.finditer('pk', 'th thpkd is pk dsspk')

for m in itr:
    print(m.start(), "-" ,m.end(), "-->",m.group())
# 5 - 7 --> pk
# 12 - 14 --> pk
# 18 - 20 --> pk

itr = re.finditer('pk', 'th thpkd is pk dsspk')

print(next(itr))
# <re.Match object; span=(5, 7), match='pk'>
```

re.sub()

- re.sub(regex,replacement,targetstring, count=0, flags=0)
- sub means substitution or replacement
- In the target string every matched pattern will be replaced with provided replacement.

```
print(re.sub(r'\d+', '#', 'foo.123.bar.789.baz'))
# 'foo.#.bar.#.baz'

print(re.sub(r'\w+', 'xxx', 'foo.bar.baz.qux', count=2))
# 'xxx.xxx.baz.qux'
```

re.subn()

- `re.subn(<regex>, <repl>, <string>, count=0, flags=0)`

It is exactly same as `sub` except it can also returns the number of replacements.

This function returns a tuple where first element is result string and second element is number of replacements.


```
print(re.subn(r'\w+', 'xxx', 'foo.bar.baz.qux'))
# ('xxx.xxx.xxx.xxx', 4)
```

re.split()

- `re.split(<regex>, <string>, maxsplit=0, flags=0)`
- Splits a string into substrings.

```
print(re.split('\s*[;,/\]\s*', 'foo,bar ; baz / qux'))
# ['foo', 'bar', 'baz', 'qux']

print(re.split(r'\s*', 'foo, bar, baz, qux, quux, corge', maxsplit=3))
# ['foo', 'bar', 'baz', 'qux, quux, corge']
```



re.escape()

- Escapes characters in a regex

```
print(re.match(re.escape('foo^bar(baz)|qux'), 'foo^bar(baz)|qux'))
# <re.Match object; span=(0, 16), match='foo^bar(baz)|qux'>
```

4.4 Metacharacters

Character(s)	Meaning
.	Matches any single character except newline(\n)
^	Anchors a match at the start of a string Complements a character class
\$	Anchors a match at the end of a string
*	Matches zero or more repetitions
+	Matches one or more repetitions
?	Matches zero or one repetition, Specifies the non-greedy versions of *, +, and ?, Introduces a lookahead or lookbehind assertion, Creates a named group
{ }	Matches an explicitly specified number of repetitions
\	Escapes a metacharacter of its special meaning, Introduces a special character class. Introduces a grouping backreference
[]	Specifies a character class
`	`
()	Creates a group
., #, =, !	Designate a specialized group
<>	Creates a named group

Character classes

1. [abc]
 - Either a or b or c
2. [^abc]
 - Except a and b and c
3. [a-z]
 - Any Lower case alphabet symbol
4. [A-Z]
 - Any upper case alphabet symbol
5. [a-zA-Z]
 - Any alphabet symbol
6. [0-9]

- Any digit from 0 to 9
- 7. `[a-zA-Z0-9]`
 - Any alphanumeric character
- 8. `^[a-zA-Z0-9]`
 - Except alphanumeric characters(Special Characters)
- 9. `ba[artz]`
 - 'ba' followed by any of 'a', 'r', 't', 'z'

```
# 3 consecutive decimal digit
print(re.search('[0-9][0-9][0-5]',string))
# <re.Match object; span=(3, 6), match='123'>

print(re.search('[0-9][0-9][0-5]', 'z999ac'))
# None
```

```
print(re.search('ba[artz]', 'foobarqux'))
# <re.Match object; span=(3, 6), match='bar'>
```

```
# matches any hexadecimal digit character

print(re.search('[0-9a-fA-f]', '--- a0 ---'))
# <re.Match object; span=(4, 5), match='a'>
```

Pre defined Character classes

- `\s`
 - Space character
- `\S`
 - Any character except space character
- `\d`
 - Essentially shorthand for `[0-9]`
- `\D`
 - Essentially shorthand for `^[^0-9]`
- `\w`
 - Essentially shorthand for `[a-zA-Z0-9_]`
- `\W`
 - Essentially shorthand for `^[^a-zA-Z0-9_]`
- `.`
 - Any character including special characters


```
# digit any digit
print(re.search('[0-9].[0-5]', 'w4s4adz'))
# <re.Match object; span=(1, 4), match='4s4'>
```

```
print(re.search('\w', '#(.a$@&'))
# <re.Match object; span=(3, 4), match='a'>

print(re.search('\w', '#(.a$@&'))
# <re.Match object; span=(0, 1), match='#'>

print(re.search('\w', 'a_'))
# None
```

```
print(re.search('\d', 'abc4def'))
# <re.Match object; span=(3, 4), match='4'>

print(re.search('\D', '234Q678'))
# <re.Match object; span=(3, 4), match='Q'>
```

- \s and \S consider a newline to be whitespace

```
print(re.search('\s', 'foo\nbar baz'))
# <re.Match object; span=(3, 4), match='\n'>

print(re.search('\S', ' \n foo \n '))
# <re.Match object; span=(4, 5), match='f'>
```

Quantifiers

- a
 - Exactly one 'a'
- a+
 - Atleast one 'a'
- a*
 - Any number of a's including zero number
- a?
 - Atmost one 'a' ie either zero number or one number
- a{m}
 - Exactly m number of a's

- $a\{m,n\}$
 - Minimum m number of a's and Maximum n number of a's

*

```
print(re.search('foo-*bar', 'foo---bar'))
# <re.Match object; span=(0, 9), match='foo---bar'>

print(re.search('foo.*bar', '# foo $qux@grault % bar #'))
# <re.Match object; span=(2, 23), match='foo $qux@grault % bar'>
```

Greedy (.*)

```
print(re.search('<.*>', '%<foo> <bar> <baz>%'))
# <re.Match object; span=(1, 18), match='<foo> <bar> <baz>'>
```

+

```
print(re.search('foo-+bar', 'foobar'))
# None

print(re.search('foo-+bar', 'foo---bar'))
# <re.Match object; span=(0, 9), match='foo---bar'>
```

Greedy (.+)

```
print(re.search('<.+>', '%<foo> <bar> <baz>%'))
# <re.Match object; span=(1, 18), match='<foo> <bar> <baz>'>
```

?

```
print(re.search('foo-?bar', 'foobar'))
# <re.Match object; span=(0, 6), match='foobar'>

print(re.search('foo-?bar', 'foo--bar'))
# None
```

Non-greedy (.*)

```
print(re.search('<.*?>', '%<foo> <bar> <baz>%'))
# <re.Match object; span=(1, 6), match='<foo>'>
```

{m}

```
print(re.search('x-{3}x', 'x---x'))
# <re.Match object; span=(0, 5), match='x---x'>
```

{m,n}

- <regex>{m,n}
 - Matches m to n no of repetition
- <regex>{,n} or <regex>{0,n}
 - Matches any to n no of repetition
- <regex>{m,}
 - Matches m to any no of repetition
- <regex>{,} or <regex>{0,} or <regex>{*}
 - Matches any no of repetition

```
print(re.search('x-{1,3}x', 'x---x'))
# <re.Match object; span=(0, 5), match='x---x'>

print(re.search('x-{1,3}x', 'x----x'))
# None
```

Boundary Check

- `^x` or `\A`
 - It will check whether target string starts with x or not
- `x$` or `\Z`
 - It will check whether target string ends with x or not
- `\b`
 - Match a word boundary consist of `[a-zA-Z0-9_]`
 - Other than `[a-zA-Z0-9_]` are as boundary, then its a match
- `\B`
 - Anchors a match to a location that isn't a word boundary

```
print(re.search('^po', 'pool'))
# <re.Match object; span=(0, 2), match='po'>

print(re.search('\Apo', 'olpo'))
# None
```

```
print(re.search('po\Z', 'olpo'))
# <re.Match object; span=(2, 4), match='po'>

print(re.search('po\Z', 'olpo\n'))
# A special case use ` $ ` (but not ` \Z `)

print(re.search('po$', 'olpo\n'))
# <re.Match object; span=(2, 4), match='po'>
```

```
# At START
print(re.search(r'\bbar', 'foo bar'))
# <re.Match object; span=(4, 7), match='bar'>

print(re.search(r'\b123', '_123'))
# None

print(re.search(r'\b123', '#123'))
#<re.Match object; span=(1, 4), match='123'>

print(re.search(r'\b#123', '#123'))
# None

print(re.search(r'\b123#', '#123'))
# None

print(re.search(r'\b123#', '#123#'))
# <re.Match object; span=(1, 5), match='123#'>

# At END
print(re.search(r'bar\b', 'foobar'))
# <re.Match object; span=(3, 6), match='bar'>
```

```
print(re.search(r'\Bfoo\B', 'zyxfooxyz'))
# <re.Match object; span=(3, 6), match='foo'>

print(re.search(r'\Bfoo', 'zyxfooxyz'))
# <re.Match object; span=(3, 6), match='foo'>
```

More

- \ backslash
 - Removes the special meaning of a metacharacter.

```
print(re.search('\.', 'foo.bar'))
# <re.Match object; span=(3, 4), match='.'>

print(re.search(r'\\', r'foo\bar'))
# <re.Match object; span=(3, 4), match='\\'>

print(re.search('\\\\', r'foo\bar'))
# <re.Match object; span=(3, 4), match='\\'>
```

4.5 Advance RegEx

() - grouping

```
print(re.search('(bar)+', 'foo barbar baz'))
# <re.Match object; span=(4, 10), match='barbar'>
```

- `m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz')`
 - 3 groups are used
 - `m` is `<re.Match object; span=(0, 12), match='foo,quux,baz'>`
 - `m.group()`
 - `'foo,quux,baz'`
 - `m.group(3)`
 - `'baz'`
 - `m.groups()`
 - `('foo', 'quux', 'baz')`

```
m1 = re.search('\w+, \w+, \w+', 'foo, quux, baz')
print(m1.groups())
# ()
```

Lookahead and Lookbehind Assertions

`?= <lookahead_regex>`

- Creates a positive lookahead assertion.

```
print(re.search('foo(?=[a-z])', 'foobar'))  
# <re.Match object; span=(0, 3), match='foo'>  
  
print(re.search('foo(?=[a-z])', 'foo123'))  
# None
```

?! <lookahead_regex>

- Creates a negative lookahead assertion.

```
print(re.search('foo(?![a-z])', 'foobar'))  
# None
```

?<= <lookbehind_regex>

- Creates a positive lookbehind assertion

```
print(re.search('(?!=[a-z])bar', 'foobar'))  
# <_sre.SRE_Match object; span=(3, 6), match='bar'>
```

?<! <lookbehind_regex>

- Creates a negative lookbehind assertion.

```
print(re.search('(?!qux)bar', 'foobar'))  
# <_sre.SRE_Match object; span=(3, 6), match='bar'>
```

Miscellaneous Metacharacters

Vertical bar, or pipe (|)

- Specifies a set of alternatives on which to match.

```
print(re.search('foo|bar|baz', 'bar'))  
# <re.Match object; span=(0, 3), match='bar'>
```

re.I or re.IGNORECASE

- Makes matching case insensitive.

```
print(re.search('a+', 'aaaAAA', re.I))  
# <re.Match object; span=(0, 6), match='aaaAAA'>
```

Reference

- <https://realpython.com/regex-python/> (<https://realpython.com/regex-python/>)
-