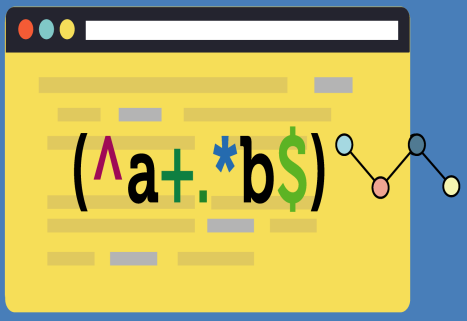
Scientific Programming

PYTHON REGEX

A Little Guide



Scientific Programmer

Python REGEX

A Little Guide

Scientific Programmer

This book is for sale at http://leanpub.com/pythonregex

This version was published on 2018-09-09

Contents

Python RegEx

Hello coders! Let's start our quest with regular expressions (RegEx). In Python, the module `re` provides full support for Perl-like regular expressions in Python. We need to remember that there are many characters in Python, which would have special meaning when they are used in regular expression. To avoid bugs while dealing with regular expressions, we use raw strings as `r'expression'`.

The re module in Python provides multiple methods to perform queries on an input string. Here are the most commonly used methods:

- `re.match()`
- `re.search()`
- `re.split()`
- `re.sub()`
- `re.findall()`
- `re.compile()`

We will look at these function and related flags with examples in the next section.

Python Regular Expression Patterns List

The following table lists the regular expression syntax that is available in Python. Note that any Regex can be concatenated to form new regular expressions; if `X` and `Y` are both regular expressions, then `XY` is also a regular expression.

Pattern	Description
`.`	Matches any single character except newline. Using `m` option allows it to match newline as well.
`^`	Matches the start of the string, and in `re.MULTILINE` (see the next lesson on how to change to multiline) mode also matches immediately after each newline.
`$`	Matches end of line. In `re.MULTILINE` mode also matches before a newline.
`[.]`	Matches any single character in brackets.
`[^.]`	Matches any single character not in brackets.
`*`	Matches 0 or more occurrences of preceding expression.
`+`	Matches 1 or more occurrence of preceding expression.
`?`	Matches 0 or 1 occurrence of preceding expression.
`{n}`	Matches exactly `n` number of occurrences of preceding expression.
`{n,}`	Matches `n` or more occurrences of preceding expression.
`{n, m}`	Matches at least `n` and at most `m` occurrences of preceding expression. For example, `x{3,5}` will match from `3` to `5` `'x'` characters.

Pattern	Description
`x y`	Matches either `x` or `y`.
`\d`	Matches digits. Equivalent to `[0-9]`.
`\D`	Matches nondigits.
`\w`	Matches word characters.
`\W`	Matches nonword characters.
`\z`	Matches end of string.
`\G`	Matches point where last match finished.
`\b`	Matches the empty string, but only at the beginning or end of a word. Boundary between word and non-word and `/B` is opposite of `/b`. Example `r"\btwo\b"` for searching two from `'one two three'`.

Pattern	Description
`\B`	Matches nonword boundaries.
`\n, \t`	Matches newlines, carriage returns, tabs, etc.
`\s`	Matches whitespace.
`\S`	Matches nonwhitespace.
`\A`	Matches beginning of string.
`\Z`	Matches end of string. If a newline exists, it matches just before newline.

Groups and Lookarounds

More details later:

Pattern	Description
`(re)`	Groups regular expressions and remembers matched text.
`(?: re)`	Groups regular expressions without remembering matched text. For example, the expression `(?:x{6})*` matches any multiple of six 'x' characters.
`(?#...)`	Comment.
`(?= ...)`	Matches if ... matches next, but doesn't consume any of the string. This is called a **lookahead assertion**. For example, `Scientific (?=Python)` will match `Scientific` only if it's followed by `Python`.
`(?!...)`	Matches if ... doesn't match next. This is a negative lookahead assertion.
`(?<=...)`	Matches if the current position in the string is preceded by a match for ... that ends at the current position.

Python regex `match` function

The `match` function attempts to match a `re` pattern to string with optional flags.

Here is the syntax for this function âˆ’

```
re.match(pattern, string, flags=0)
```

Where,

- `pattern` is the regular expression to be matched,
- `string` is the string to be searched to match the pattern at the beginning of string and
- `flags`, which you can specify different flags using bitwise OR (`|`).

Match Flags

Modifier	Description
`re.I`	Performs case-insensitive matching.
`re.L`	Interprets words according to the current locale. This interpretation affects the alphabetic group (`\w` and `\W`), as well as word boundary behavior (`\b` and `\B`).
`re.M`	Makes `$` match the end of a line and makes `^` match the start of any line.
`re.S`	Makes a period (dot) match any character, including a newline.
`re.U`	Interprets letters according to the Unicode character set. This flag affects the behavior of `\w`, `\W`, `\b`, `\B`.
`re.X`	It ignores whitespace (except inside a set `[]` or when escaped by a `backslash` and treats unescaped `#` as a comment marker.

Return values

- The `re.match` function returns a `match object` on **success** and `None` upon failure. -
- Use `group(n)` or `groups()` function of match object to get matched expression, e.g., `group(n=0)` returns entire match (or specific subgroup `n`)
- The function `groups()` returns all matching subgroups in a tuple (empty if there weren't any).

Example 1

Let's find the words before and after the word `to`:

```
1  #!/usr/bin/python
2  import re
3
4  line = "Learn to Analyze Data with Scientific Python";
5
6  m = re.match( r'(.*) to (.*?) .*', line, re.M|re.I)
7
8  if m:
9     print "m.group() : ", m.group()
10    print "m.group(1) : ", m.group(1)
11    print "m.group(2) : ", m.group(2)
12 else:
13    print "No match!!"
```

The first group `(.*)` identified the string: Learn and the next group `(*.?)` identified the string: **Analyze**. Output:

```
m.group() :  Learn to Analyze Data with Scientific Python
m.group(1) :  Learn
m.group(2) :  Analyze
```

Example 2

`groups([default])` returns a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern.

```
#!/usr/bin/python
import re

line = "Learn Data, Python";

m = re.match( r'(\w+) (\w+)', line, re.M|re.I)

if m:
   print "m.group() : ", m.groups()
   print "m.group (1,2)", m.group(1, 2)
else:
   print "No match!!"
```

Output:

```
m.group() :  ('Learn', 'Data')
m.group (1,2) ('Learn', 'Data')
```

Example 3

`groupdict([default])` returns a dictionary containing all the named subgroups of the match, keyed by the subgroup name.

```
#!/usr/bin/python
import re

number = "124.13";

m = re.match( r'(?P<Expotent>\d+)\.(?P<Fraction>\d+)', nu\
mber)

if m:
   print "m.groupdict() : ", m.groupdict()
else:
   print "No match!!"
```

Output: `m.groupdict() : {'Expotent': '124', 'Fraction': '13'}`

Example 4

Start, end. How can we match the start or end of a string? We can use the "A" and "Z" metacharacters. We precede them with a backslash. We match strings that start with a certain letter, and those that end with another.

```
import re

values = ["Learn", "Live", "Python"];

for value in values:
    # Match the start of a string.
    result = re.match("\AL.+", value)
    if result:
        print("START MATCH [L]:", value)

    # Match the end of a string.
    result2 = re.match(".+n\Z", value)
```

```
    if result2:
        print("END MATCH [n]:", value)
```

Output:

```
output

('START MATCH [L]:', 'Learn')
('END MATCH [n]:', 'Learn')
('START MATCH [L]:', 'Live')
('END MATCH [n]:', 'Python')
```

Example 5

`start([group])` and `end([group])` return the indices of the start and end of the substring matched by `group`. See the next lesson for an example.