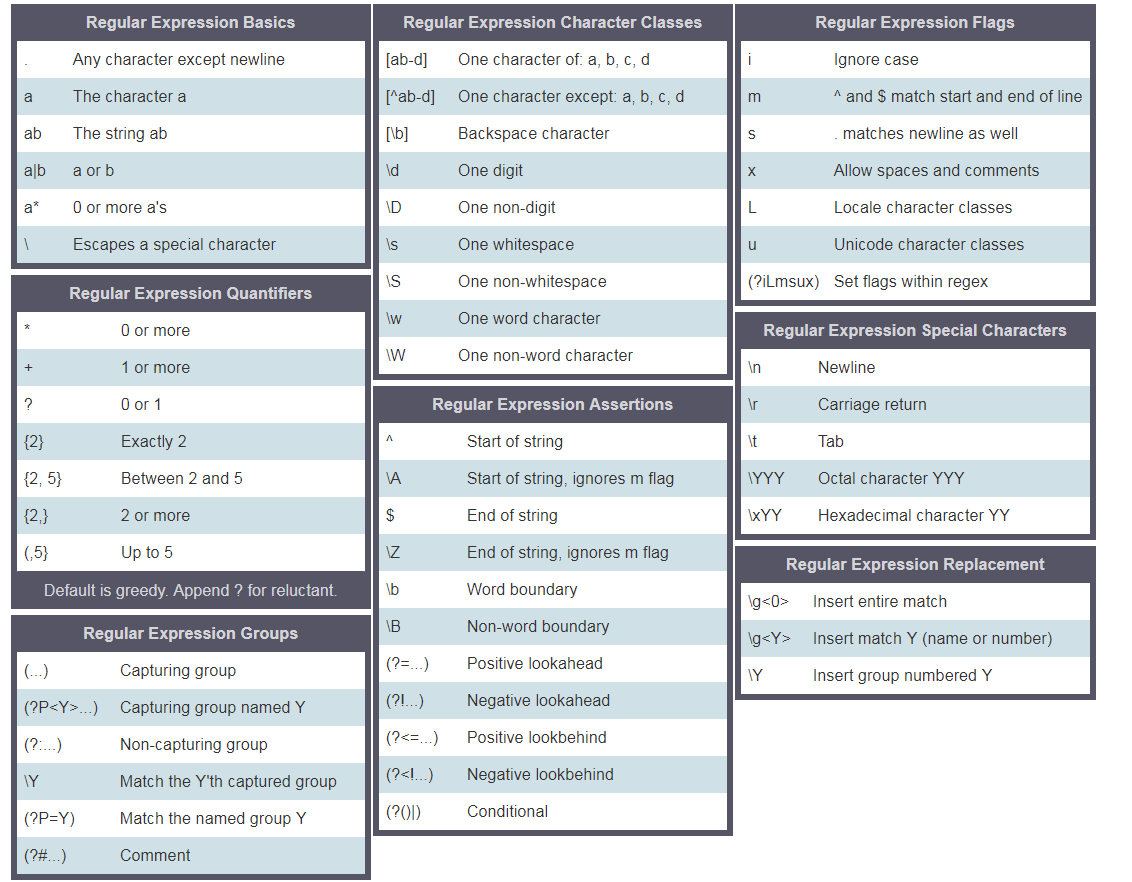
Python - Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.



## The *match* Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

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

Here is the description of the parameters −

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **pattern**  This is the regular expression to be matched. |
| 2 | **string**  This is the string, which would be searched to match the pattern at the beginning of string. |
| 3 | **Flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

The *re.match* function returns a **match** object on success, **None** on failure. We use*group(num)* or *groups()*function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **Sr.No.** | **Match Object Method & Description** |
| 1 | **group(num=0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

### Example

#!/usr/bin/python

import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if matchObj:

print "matchObj.group() : ", matchObj.group()

print "matchObj.group(1) : ", matchObj.group(1)

print "matchObj.group(2) : ", matchObj.group(2)

else:

print "No match!!"

When the above code is executed, it produces following result −

matchObj.group() : Cats are smarter than dogs

matchObj.group(1) : Cats

matchObj.group(2) : smarter

## What are various methods of Regular Expressions?

The ‘re’ package provides multiple methods to perform queries on an input string. Here are the most commonly used methods, I will discuss:

1. re.match()
2. re.search()
3. re.findall()
4. re.split()
5. re.sub()
6. re.compile()

### re.match(pattern, string):

This method finds match if it occurs at start of the string. For example, calling match() on the string ‘AV Analytics AV’ and looking for a pattern ‘AV’ will match. However, if we look for only Analytics, the pattern will not match. Let’s perform it in python now.

Code

import re

result = re.match(r'AV', 'AV Analytics Vidhya AV')

print result

**Output:**

<\_sre.SRE\_Match object at 0x0000000009BE4370>

Above, it shows that pattern match has been found. To print the matching string we’ll use method group (It helps to return the matching string). Use “r” at the start of the pattern string, it designates a python raw string.

result = re.match(r'AV', 'AV Analytics Vidhya AV')

print result.group(0)

**Output:**

AV

Let’s now find ‘Analytics’ in the given string. Here we see that string is not starting with ‘AV’ so it should return no match. Let’s see what we get:

Code

result = re.match(r'Analytics', 'AV Analytics Vidhya AV')

print result

**Output:**

None

There are methods like start() and end() to know the start and end position of matching pattern in the string.

Code

result = re.match(r'AV', 'AV Analytics Vidhya AV')

print result.start()

print result.end()

**Output:**

0

2

Above you can see that start and end position of matching pattern ‘AV’ in the string and sometime it helps a lot while performing manipulation with the string.

### re.search(pattern, string):

It is similar to match() but it doesn’t restrict us to find matches at the beginning of the string only. Unlike previous method, here searching for pattern ‘Analytics’ will return a match.

Code

result = re.search(r'Analytics', 'AV Analytics Vidhya AV')

print result.group(0)

**Output:**

Analytics

Here you can see that, search() method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern.

### re.findall (pattern, string):

Ithelps to get a list of all matching patterns. It has no constraints of searching from start or end. If we will use method findall to search ‘AV’ in given string it will return both occurrence of AV. While searching a string, I would recommend you to use **re.findall()**always, it can work like re.search() and re.match() both.

Code

result = re.findall(r'AV', 'AV Analytics Vidhya AV')

print result

**Output:**

['AV', 'AV']

### re.split(pattern, string, [maxsplit=0]):

This methods helps to split string by the occurrences of given pattern.

Code

result=re.split(r'y','Analytics')

result

**Output:**

['Anal', 'tics']

Above, we have split the string “Analytics” by “y”. Method split() has another argument “**maxsplit**“. It has default value of zero. In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string. Let’s look at the example below:

Code

result=re.split(r'i','Analytics Vidhya')

print result

**Output**:

['Analyt', 'cs V', 'dhya'] #It has performed all the splits that can be done by pattern "i".

Code

result=re.split(r'i','Analytics Vidhya',maxsplit=1)

result

**Output**:

['Analyt', 'cs Vidhya']

Here, you can notice that we have fixed the maxsplit to 1. And the result is, it has only two values whereas first example has three values.

### re.sub(pattern, repl, string):

It helps to search a pattern and replace with a new sub string. If the pattern is not found, string is returned unchanged.

Code

result=re.sub(r'India','the World','AV is largest Analytics community of India')

result

**Output:**

'AV is largest Analytics community of the World'

### re.compile(pattern, repl, string):

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

Code

import re

pattern=re.compile('AV')

result=pattern.findall('AV Analytics Vidhya AV')

print result

result2=pattern.findall('AV is largest analytics community of India')

print result2

**Output:**

['AV', 'AV']

['AV']

### Quick Recap of various methods:

Till now,  we looked at various methods of regular expression using a constant pattern (fixed characters). But, what if we do not have a constant search pattern and we want to return specific set of characters (defined by a rule) from a string?  Don’t be intimidated.

This can easily be solved by defining an expression with the help of pattern operators (meta  and literal characters). Let’s look at the most common pattern operators.

### What are the most commonly used operators?

Regular expressions can specify patterns, not just fixed characters. Here are the most commonly used operators that helps to generate an expression to represent required characters in a string or file. It is commonly used in web scrapping and  text mining to extract required information.

|  |  |
| --- | --- |
| **Operators** | **Description** |
| . | Matches with any single character except newline ‘\n’. |
| ? | match 0 or 1 occurrence of the pattern to its left |
| + | 1 or more occurrences of the pattern to its left |
| \* | 0 or more occurrences of the pattern to its left |
| \w | Matches with a alphanumeric character whereas \W (upper case W) matches non alphanumeric character. |
| \d | Matches with digits [0-9] and /D (upper case D) matches with non-digits. |
| \s | Matches with a single white space character (space, newline, return, tab, form) and \S (upper case S) matches any non-white space character. |
| \b | boundary between word and non-word and /B is opposite of /b |
| [..] | Matches any single character in a square bracket and [^..] matches any single character not in square bracket |
| \ | It is used for special meaning characters like \. to match a period or \+ for plus sign. |
| ^ and $ | ^ and $ match the start or end of the string respectively |
| {n,m} | Matches at least n and at most m occurrences of preceding expression if we write it as {,m} then it will return at least any minimum occurrence to max m preceding expression. |
| a| b | Matches either a or b |
| ( ) | Groups regular expressions and returns matched text |
| \t, \n, \r | Matches tab, newline, return |

For more details on  meta characters “(“, “)”,”|” and others details , you can refer this link (<https://docs.python.org/2/library/re.html>).

Now, let’s understand the pattern operators by looking at the below examples.

## Some Examples of Regular Expressions

### Problem 1: Return the first word of a given string

**Solution-1**Extract each character**(**using “**\w**“**)**

Code

import re

result=re.findall(r'.','AV is largest Analytics community of India')

print result

**Output:**

['A', 'V', ' ', 'i', 's', ' ', 'l', 'a', 'r', 'g', 'e', 's', 't', ' ', 'A', 'n', 'a', 'l', 'y', 't', 'i', 'c', 's', ' ', 'c', 'o', 'm', 'm', 'u', 'n', 'i', 't', 'y', ' ', 'o', 'f', ' ', 'I', 'n', 'd', 'i', 'a']

Above, space is also extracted, now to avoid it use “**\w**” instead of “**.**“.

Code

result=re.findall(r'\w','AV is largest Analytics community of India')

print result

**Output:**

['A', 'V', 'i', 's', 'l', 'a', 'r', 'g', 'e', 's', 't', 'A', 'n', 'a', 'l', 'y', 't', 'i', 'c', 's', 'c', 'o', 'm', 'm', 'u', 'n', 'i', 't', 'y', 'o', 'f', 'I', 'n', 'd', 'i', 'a']

**Solution-2**Extract each word**(**using “**\***” or “**+**“**)**

Code

result=re.findall(r'\w\*','AV is largest Analytics community of India')

print result

**Output:**

['AV', '', 'is', '', 'largest', '', 'Analytics', '', 'community', '', 'of', '', 'India', '']

Again, it is returning space as a word because “**\***” returns zero or more matches of pattern to its left. Now to remove spaces we will go with “**+**“.

Code

result=re.findall(r'\w+','AV is largest Analytics community of India')

print result

**Output:**

['AV', 'is', 'largest', 'Analytics', 'community', 'of', 'India']

**Solution-3**Extract each word**(**using “**^**“**)**

Code

result=re.findall(r'^\w+','AV is largest Analytics community of India')

print result

**Output:**

['AV']

If we will use “$” instead of “^”, it will return the word from the end of the string. Let’s look at it.

Code

result=re.findall(r'\w+$','AV is largest Analytics community of India')

print result

**Output:**

[‘India’]

### Problem 2: Return the first two character of each word

**Solution-1**Extract consecutive two characters of each word, excluding spaces (using “**\w**“**)**

Code

result=re.findall(r'\w\w','AV is largest Analytics community of India')

print result

Output:

['AV', 'is', 'la', 'rg', 'es', 'An', 'al', 'yt', 'ic', 'co', 'mm', 'un', 'it', 'of', 'In', 'di']

**Solution-2**Extract consecutive two characters those available at start of word boundary (using “**\b**“**)**

Code

result=re.findall(r'\b\w.','AV is largest Analytics community of India')

print result

**Output:**

['AV', 'is', 'la', 'An', 'co', 'of', 'In']

### Problem 3: Return the domain type of given email-ids

To explain it in simple manner, I will again go with a stepwise approach:

**Solution-1**Extract all characters after “@”

Code

result=re.findall(r'@\w+','abc.test@gmail.com, xyz@test.in, test.first@analyticsvidhya.com, first.test@rest.biz')

print result

**Output:** ['@gmail', '@test', '@analyticsvidhya', '@rest']

Above, you can see that “.com”, “.in” part is not extracted. To add it, we will go with below code.

result=re.findall(r'@\w+.\w+','abc.test@gmail.com, xyz@test.in, test.first@analyticsvidhya.com, first.test@rest.biz')

print result

**Output:**

['@gmail.com', '@test.in', '@analyticsvidhya.com', '@rest.biz']

**Solution – 2**Extract only domain name using “( )”

Code

result=re.findall(r'@\w+.(\w+)','abc.test@gmail.com, xyz@test.in, test.first@analyticsvidhya.com, first.test@rest.biz')

print result

**Output:**

['com', 'in', 'com', 'biz']

### Problem 4: Return date from given string

Here we will use “**\d**” to extract digit.

**Solution:**

Code

result=re.findall(r'\d{2}-\d{2}-\d{4}','Amit 34-3456 12-05-2007, XYZ 56-4532 11-11-2011, ABC 67-8945 12-01-2009')

print result

**Output:**

['12-05-2007', '11-11-2011', '12-01-2009']

If you want to extract only year again parenthesis “( )” will help you.

Code

result=re.findall(r'\d{2}-\d{2}-(\d{4})','Amit 34-3456 12-05-2007, XYZ 56-4532 11-11-2011, ABC 67-8945 12-01-2009')

print result

**Output:**

['2007', '2011', '2009']

### Problem 5: Return all words of a string those starts with vowel

**Solution-1**Return each words

Code

result=re.findall(r'\w+','AV is largest Analytics community of India')

print result

**Output:**

['AV', 'is', 'largest', 'Analytics', 'community', 'of', 'India']

**Solution-2**Return words starts with alphabets (using [])

Code

result=re.findall(r'[aeiouAEIOU]\w+','AV is largest Analytics community of India')

print result

**Output:**

['AV', 'is', 'argest', 'Analytics', 'ommunity', 'of', 'India']

Above you can see that it has returned “argest” and “ommunity” from the mid of words. To drop these two, we need to use “\b” for word boundary.

**Solution- 3**

Code

result=re.findall(r'\b[aeiouAEIOU]\w+','AV is largest Analytics community of India')

print result

**Output:**

['AV', 'is', 'Analytics', 'of', 'India']

In similar ways, we can extract words those starts with constant using “^” within square bracket.

Code

result=re.findall(r'\b[^aeiouAEIOU]\w+','AV is largest Analytics community of India')

print result

**Output:**

[' is', ' largest', ' Analytics', ' community', ' of', ' India']

Above you can see that it has returned words starting with space. To drop it from output, include space in square bracket[].

Code

result=re.findall(r'\b[^aeiouAEIOU ]\w+','AV is largest Analytics community of India')

print result

**Output:**

['largest', 'community']

### Problem 6: Validate a phone number (phone number must be of 10 digits and starts with 8 or 9)

We have a list phone numbers in list “li” and here we will validate phone numbers using regular

**Solution**

Code

import re

li=['9999999999','999999-999','99999x9999']

for val in li:

if re.match(r'[8-9]{1}[0-9]{9}',val) and len(val) == 10:

print 'yes'

else:

print 'no'

**Output:**

yes

no

no

### Problem 7: Split a string with multiple delimiters

**Solution**

Code

import re

line = 'asdf fjdk;afed,fjek,asdf,foo' # String has multiple delimiters (";",","," ").

result= re.split(r'[;,\s]', line)

print result

**Output:**

['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']

We can also use method **re.sub()**to replace these multiple delimiters with one as space ” “.

Code

import re

line = 'asdf fjdk;afed,fjek,asdf,foo'

result= re.sub(r'[;,\s]',' ', line)

print result

**Output:**

asdf fjdk afed fjek asdf foo

### Problem 8: Retrieve Information from HTML file

I want to extract information from a HTML file (see below sample data). Here we need to extract information available between <td> and </td> except the first numerical index. I have assumed here that below html code is stored in a string **str**.

Sample HTML file (str)

<tr align="center"><td>1</td> <td>Noah</td> <td>Emma</td></tr>

<tr align="center"><td>2</td> <td>Liam</td> <td>Olivia</td></tr>

<tr align="center"><td>3</td> <td>Mason</td> <td>Sophia</td></tr>

<tr align="center"><td>4</td> <td>Jacob</td> <td>Isabella</td></tr>

<tr align="center"><td>5</td> <td>William</td> <td>Ava</td></tr>

<tr align="center"><td>6</td> <td>Ethan</td> <td>Mia</td></tr>

<tr align="center"><td>7</td> <td HTML>Michael</td> <td>Emily</td></tr>

**Solution:**

Code

result=re.findall(r'<td>\w+</td>\s<td>(\w+)</td>\s<td>(\w+)</td>',str)

print result

**Output:**

[('Noah', 'Emma'), ('Liam', 'Olivia'), ('Mason', 'Sophia'), ('Jacob', 'Isabella'), ('William', 'Ava'), ('Ethan', 'Mia'), ('Michael', 'Emily')]

You can read html file using library urllib2 (see below code).

Code

import urllib2

response = urllib2.urlopen('')

html = response.read()

## The *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function −

re.search(pattern, string, flags=0)

Here is the description of the parameters −

|  |  |
| --- | --- |
| **Sr.No.** | **Parameter & Description** |
| 1 | **pattern**  This is the regular expression to be matched. |
| 2 | **string**  This is the string, which would be searched to match the pattern anywhere in the string. |
| 3 | **flags**  You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below. |

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)*or *groups()*function of **match** object to get matched expression.

|  |  |
| --- | --- |
| **Sr.No.** | **Match Object Methods & Description** |
| 1 | **group(num=0)**  This method returns entire match (or specific subgroup num) |
| 2 | **groups()**  This method returns all matching subgroups in a tuple (empty if there weren't any) |

### Example

#!/usr/bin/python

import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.\*) are (.\*?) .\*', line, re.M|re.I)

if searchObj:

print "searchObj.group() : ", searchObj.group()

print "searchObj.group(1) : ", searchObj.group(1)

print "searchObj.group(2) : ", searchObj.group(2)

else:

print "Nothing found!!"

When the above code is executed, it produces following result −

searchObj.group() : Cats are smarter than dogs

searchObj.group(1) : Cats

searchObj.group(2) : smarter

## Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search**checks for a match anywhere in the string (this is what Perl does by default).

### Example

#!/usr/bin/python

import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)

if matchObj:

print "match --> matchObj.group() : ", matchObj.group()

else:

print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)

if searchObj:

print "search --> searchObj.group() : ", searchObj.group()

else:

print "Nothing found!!"

When the above code is executed, it produces the following result −

No match!!

search --> matchObj.group() : dogs

## Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

### Syntax

re.sub(pattern, repl, string, max=0)

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

### Example

#!/usr/bin/python

import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments

num = re.sub(r'#.\*$', "", phone)

print "Phone Num : ", num

# Remove anything other than digits

num = re.sub(r'\D', "", phone)

print "Phone Num : ", num

When the above code is executed, it produces the following result −

Phone Num : 2004-959-559

Phone Num : 2004959559

## Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

|  |  |
| --- | --- |
| **Sr.No.** | **Modifier & Description** |
| 1 | **re.I**  Performs case-insensitive matching. |
| 2 | **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). |
| 3 | **re.M**  Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string). |
| 4 | **re.S**  Makes a period (dot) match any character, including a newline. |
| 5 | **re.U**  Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B. |
| 6 | **re.X**  Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker. |

## Regular Expression Patterns

Except for control characters, **(+ ? . \* ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python −

|  |  |
| --- | --- |
| **Sr.No.** | **Pattern & Description** |
| 1 | **^**  Matches beginning of line. |
| 2 | **$**  Matches end of line. |
| 3 | **.**  Matches any single character except newline. Using m option allows it to match newline as well. |
| 4 | **[...]**  Matches any single character in brackets. |
| 5 | **[^...]**  Matches any single character not in brackets |
| 6 | **re\***  Matches 0 or more occurrences of preceding expression. |
| 7 | **re+**  Matches 1 or more occurrence of preceding expression. |
| 8 | **re?**  Matches 0 or 1 occurrence of preceding expression. |
| 9 | **re{ n}**  Matches exactly n number of occurrences of preceding expression. |
| 10 | **re{ n,}**  Matches n or more occurrences of preceding expression. |
| 11 | **re{ n, m}**  Matches at least n and at most m occurrences of preceding expression. |
| 12 | **a| b**  Matches either a or b. |
| 13 | **(re)**  Groups regular expressions and remembers matched text. |
| 14 | **(?imx)**  Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| 15 | **(?-imx)**  Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected. |
| 16 | **(?: re)**  Groups regular expressions without remembering matched text. |
| 17 | **(?imx: re)**  Temporarily toggles on i, m, or x options within parentheses. |
| 18 | **(?-imx: re)**  Temporarily toggles off i, m, or x options within parentheses. |
| 19 | **(?#...)**  Comment. |
| 20 | **(?= re)**  Specifies position using a pattern. Doesn't have a range. |
| 21 | **(?! re)**  Specifies position using pattern negation. Doesn't have a range. |
| 22 | **(?> re)**  Matches independent pattern without backtracking. |
| 23 | **\w**  Matches word characters. |
| 24 | **\W**  Matches nonword characters. |
| 25 | **\s**  Matches whitespace. Equivalent to [\t\n\r\f]. |
| 26 | **\S**  Matches nonwhitespace. |
| 27 | **\d**  Matches digits. Equivalent to [0-9]. |
| 28 | **\D**  Matches nondigits. |
| 29 | **\A**  Matches beginning of string. |
| 30 | **\Z**  Matches end of string. If a newline exists, it matches just before newline. |
| 31 | **\z**  Matches end of string. |
| 32 | **\G**  Matches point where last match finished. |
| 33 | **\b**  Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets. |
| 34 | **\B**  Matches nonword boundaries. |
| 35 | **\n, \t, etc.**  Matches newlines, carriage returns, tabs, etc. |
| 36 | **\1...\9**  Matches nth grouped subexpression. |
| 37 | **\10**  Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code. |

## Regular Expression Examples

### Literal characters

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **python**  Match "python". |

## Character classes

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **[Pp]ython**  Match "Python" or "python" |
| 2 | **rub[ye]**  Match "ruby" or "rube" |
| 3 | **[aeiou]**  Match any one lowercase vowel |
| 4 | **[0-9]**  Match any digit; same as [0123456789] |
| 5 | **[a-z]**  Match any lowercase ASCII letter |
| 6 | **[A-Z]**  Match any uppercase ASCII letter |
| 7 | **[a-zA-Z0-9]**  Match any of the above |
| 8 | **[^aeiou]**  Match anything other than a lowercase vowel |
| 9 | **[^0-9]**  Match anything other than a digit |

## Special Character Classes

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **.**  Match any character except newline |
| 2 | **\d**  Match a digit: [0-9] |
| 3 | **\D**  Match a nondigit: [^0-9] |
| 4 | **\s**  Match a whitespace character: [ \t\r\n\f] |
| 5 | **\S**  Match nonwhitespace: [^ \t\r\n\f] |
| 6 | **\w**  Match a single word character: [A-Za-z0-9\_] |
| 7 | **\W**  Match a nonword character: [^A-Za-z0-9\_] |

## Repetition Cases

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **ruby?**  Match "rub" or "ruby": the y is optional |
| 2 | **ruby\***  Match "rub" plus 0 or more ys |
| 3 | **ruby+**  Match "rub" plus 1 or more ys |
| 4 | **\d{3}**  Match exactly 3 digits |
| 5 | **\d{3,}**  Match 3 or more digits |
| 6 | **\d{3,5}**  Match 3, 4, or 5 digits |

## Nongreedy repetition

This matches the smallest number of repetitions −

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **<.\*>**  Greedy repetition: matches "<python>perl>" |
| 2 | **<.\*?>**  Nongreedy: matches "<python>" in "<python>perl>" |

## Grouping with Parentheses

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **\D\d+**  No group: + repeats \d |
| 2 | **(\D\d)+**  Grouped: + repeats \D\d pair |
| 3 | **([Pp]ython(, )?)+**  Match "Python", "Python, python, python", etc. |

## Backreferences

This matches a previously matched group again −

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **([Pp])ython&\1ails**  Match python&pails or Python&Pails |
| 2 | **(['"])[^\1]\*\1**  Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc. |

## Alternatives

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **python|perl**  Match "python" or "perl" |
| 2 | **rub(y|le))**  Match "ruby" or "ruble" |
| 3 | **Python(!+|\?)**  "Python" followed by one or more ! or one ? |

## Anchors

This needs to specify match position.

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **^Python**  Match "Python" at the start of a string or internal line |
| 2 | **Python$**  Match "Python" at the end of a string or line |
| 3 | **\APython**  Match "Python" at the start of a string |
| 4 | **Python\Z**  Match "Python" at the end of a string |
| 5 | **\bPython\b**  Match "Python" at a word boundary |
| 6 | **\brub\B**  \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone |
| 7 | **Python(?=!)**  Match "Python", if followed by an exclamation point. |
| 8 | **Python(?!!)**  Match "Python", if not followed by an exclamation point. |

## Special Syntax with Parentheses

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | **R(?#comment)**  Matches "R". All the rest is a comment |
| 2 | **R(?i)uby**  Case-insensitive while matching "uby" |
| 3 | **R(?i:uby)**  Same as above |
| 4 | **rub(?:y|le))**  Group only without creating \1 backreference |

Example: 1

IP Address Validation

aa=re.match(r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$",ip)

if aa:

ip = aa.group()

Example: 2

Email id validation

r"(^[a-zA-Z0-9\_.+-][+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$](mailto:+@[a-zA-Z0-9-]+\.%5ba-zA-Z0-9-.%5d+$))"

import re

addressToVerify ='info@emailhippo.com'

match = re.match('^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\*@[a-z0-9-]+(\.[a-z0-9-]+)\*(\.[a-z]{2,4})$', addressToVerify)

if match == None:

print('Bad Syntax')

raise ValueError('Bad Syntax')

Example : 3

Find all 3 digit numbers

>>> import re

>>> s = "88888 999999 3333 aaa 12345 hfsjkq 98765"

>>> re.findall(r"(?<!\d)\d{5}(?!\d)", s)

['88888', '12345', '98765']

import re

number =input("Please enter a number: ")

number=int(number)

while not re.match("^[0-9 \-]+$", number):

print ("Error! Make sure you only use numbers")

number = input("Please enter a number: ")

print("You picked number "+ number)

while True:

s = input("Please enter a number: ")

try:

n = int(s)

break

except ValueError:

print("Error! Make sure you only use numbers")

print("You picked number " + n)

// code contains regular expression for contact number and email address in python

str='abc@example.com'

match=re.search(r'\w+@\w+',str) #return abc@example.com

num=555-555-555

match\_num=re.search(r'^(\d{3}--\d{3}--\d{4})

,num) #return 555-555-5555