# **Module 13 -** Regular Expression

Contents

[**Module 13 -** Regular Expression 1](#_Toc216013676)

[Introduction to Regular Expression 2](#_Toc216013677)

[Metacharacters 2](#_Toc216013678)

[Reference Guide 3](#_Toc216013679)

[Regular Expression Functions 4](#_Toc216013680)

[Regular Expression Syntax 6](#_Toc216013681)

[Greedy and Non-Greedy Quantifiers 6](#_Toc216013682)

[Examples 8](#_Toc216013683)

[Reference and Additional Materials: 17](#_Toc216013684)

## 

## Introduction to Regular Expression

Regular expressions (often called regex or REs) in Python are powerful tools that can be applied in text processing, validation, web scraping, data mining, and many more real-world applications. Mastering [regex](https://docs.python.org/3/howto/regex.html) can significantly improve the efficiency of data-driven applications. [Regular expression](https://youtu.be/ovZsvN67Glc?si=wLaGs-Rf4TvHUqZF) itself is a mini programming language, but not a statement based language, but a character based programming language. It is made available through the [‘re’ module](https://docs.python.org/3/library/re.html). It is important that you import the regular expression library by using **import re**. The regular expression language is small and restricted so not all string processing tasks can be done using regex. You can use regex to specify the rules for the set of possible strings that you want to match.

## Metacharacters

[Metacharacters](https://youtu.be/j6A28L6Tmxw?si=EeZKiOsFmQr7LQ-g) are special characters that have predefined specific meanings and must be escaped with a backlash (\) to be matched literally. They are used to build complex patterns. Below in the [Reference Guide](https://media.datacamp.com/legacy/image/upload/v1665049611/Marketing/Blog/Regular_Expressions_Cheat_Sheet.pdf) is a breakdown of common metacharacters and how to handle them.

## 

## Reference Guide

* `[**^**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta4)` - Matches the beginning of the line.
* `[**$**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta5)` - Matches the end of the line.
* [`**.**`](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta3) - Matches any character (a wildcard).
* `[**\s**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_seq6)` - Matches a whitespace character.
* `[\**S**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_seq7)` - Matches a non-whitespace character (opposite of \s).
* `[**\***](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta6)` - Applies to the immediately preceding character(s) and indicates to match zero or more times.
* `**\*?**` - Applies to the immediately preceding character(s) and indicates to match zero or more times in "non-greedy mode".
* [`+`](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta7) - Applies to the immediately preceding character(s) and indicates to match one or more times.
* `**+?**` - Applies to the immediately preceding character(s) and indicates to match one or more times in "non-greedy mode".
* `[**?**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta10)` - Applies to the immediately preceding character(s) and indicates to match zero or one time.
* `**??**` - Applies to the immediately preceding character(s) and indicates to match zero or one time in "non-greedy mode".
* [‘|’](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta9) - Either or
* `[**[aeiou]**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_set1)**`** - Matches a single character as long as that character is in the specified set. In this example, it would match "a", "e", "i", "o", or "u", but no other characters.
* `[**[a-z0-9]**](https://www.w3schools.com/python/trypython.asp?)` - You can specify ranges of characters using the minus sign. This example is a single character that must be a lowercase letter or a digit.
* `[**[^A-Za-z]**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_set3)` - When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything other than an uppercase or lowercase letter.
* `**( )**` - When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using findall().
* `[**\b**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_seq2)` - Matches the empty string, but only at the start or end of a word.
* `[**\B**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_seq3)` - Matches the empty string, but not at the start or end of a word.
* `[**\d**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_seq4)` - Matches any decimal digit; equivalent to the set [0-9].
* `[**\D**](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_seq5)` - Matches any non-digit character; equivalent to the set [^0-9].
* [`**{m,n}**`](https://www.w3schools.com/python/trypython.asp?filename=demo_regex_meta8) - When curly braces are added to a regular expression, they mean there must be at least \*m\* repetitions of the preceding character(s), and at most \*n\* repetitions. Either \*m\* or \*n\* may be omitted (e.g. `"{,5}"` means at most five repetitions).

## Regular Expression Functions

All regex functions are supported by they’re module. They will be written like the following in your python script: re.findall(), re.search(), re.fullsearch(), re.split(), re.sub().

#### **re.findall()**

The [findall()](https://www.w3schools.com/python/python_regex.asp#:~:text=Try%20it%20%C2%BB-,The%20findall()%20Function,-The%20findall()) method returns all non-overlapping matches of pattern in string, as a list of strings. This is equivalent to doing find and character slicing together. It will return a list containing all matches.

import re

file\_handle = open('mbox-short.txt')

for line in file\_handle:

line = line.rstrip()

results = re.findall('\S+@\S+', line)

if len(results) > 0:

for address in results:

print(address)

#### **re.fullmatch()**

The [fullmatch() method](https://www.pythontutorial.net/python-regex/python-regex-fullmatch/) returns a match object if and only if the entire target string from the first to the last character matches the regular expression pattern. Fullmatch() is very strict in pattern matching whatever is passed inside fullmatch() parameter should be exactly matched. If the match is performed successfully it will return the entire string as a match value because we always match the entire string in fullmatch.

import re

# string length of 42

str1 = "My name is Mary and my salary is 1000$"

result = re.fullmatch(r".{38}", str1)

# print entire match object

print(result)

# print actual match value

print("Match: ", result.group())

#### **re.search()**

The [search()](https://www.w3schools.com/python/python_regex.asp#:~:text=Try%20it%20Yourself%20%C2%BB-,The%20search()%20Function,-The%20search()) method returns a match object if there is a match anywhere in the string. It is simply a smart version of the [find() method](https://www.w3schools.com/python/ref_string_find.asp) in string. The search() function returns the first location where the regular expression pattern produces a match.

## Extracting using find() method of string object

hand = open('mbox-short.txt')

for line in hand:

line = line.rstrip()

if line.find('From:') >= 0:

print(line)

#### **re.split()**

The [split()](https://www.codecademy.com/resources/docs/python/regex/split) method returns a list where the string has been split at each match. It works similarly to [Python’s built-in split()](https://www.w3schools.com/python/ref_string_split.asp), but allows for more complex patterns, such as splitting on multiple characters or specific sequences.

import re

target\_string = "Light travels at a constant speed of 1.07 billion km per hour"

# regex to split string on whitespaces

res = re.split(r"\s", target\_string)

print("All tokens:", res)

In this example, \s means wherever empty space is seen, put it as elements of a list in the result.

#### **re.sub()**

The [sub()](https://www.codecademy.com/resources/docs/python/regex/sub) method replaces one or many matches with a string. It works like a find-and-replace using regular expressions. Provide the regex pattern, the replacement string, and the original text.

import re

target\_string = "Light travels at a constant speed of 1.07 billion km per hour"

# regex for replacement

# replace space with hyphen

res = re.sub(r"\s", "-", target\_string)

# string after replacement:

print(res)

## Regular Expression Syntax

Regular Expressions are made up of both ordinary and [special characters](https://www.dataquest.io/cheat-sheet/regular-expressions-cheat-sheet/#:~:text=s)%20(%3Fu)%20(%3Fx)-,Special%20Characters,-Syntax). Ordinary characters like **A**, **a**, or **0** are the most basic elements of a regular expression because they simply match themselves. You can combine ordinary characters to form patterns, so something like one will match the exact string ‘one’.

Some characters have special meanings, like **|** and **(** for example. These special characters either represent groups or classes of characters, or how they change how nearby parts of the expression are interpreted.

[Quantifiers](https://www.pythontutorial.net/python-regex/python-regex-quantifiers/), also called repetition operators, like **\*, +**, **?** or **{m,n}**. These cannot be directly nested because doing so would create confusion, especially with [non-greedy](https://www.pythontutorial.net/python-regex/python-regex-non-greedy/) modifiers like **?**, or other modifiers in different regex engines. If you want to apply a repetition to an already-repeating pattern, you can use parentheses. For example, **(?:a{6})\*** matches any string that contains a multiple of six ‘a’ characters.

[Grouping](https://docs.python.org/3/howto/regex.html#grouping:~:text=a%20word%20boundary.-,Grouping,-%C2%B6) with parentheses () allows treating multiple characters as a single unit. This is useful when applying quantifiers to an entire section of a pattern or when [capturing](https://youtu.be/vYOeAt83C7o?si=B36Itm2-mEuW-85x) specific parts of a match for later use. Groups can be numbered automatically based on their position or assigned names for better readability.

## Greedy and Non-Greedy Quantifiers

In Python regex, [greedy](https://www.pythontutorial.net/python-regex/python-regex-greedy) quantifiers (\*, +, {m,n}) match as much text as possible before backtracking if needed. [Non-greedy](https://www.pythontutorial.net/python-regex/python-regex-non-greedy/) quantifiers (\*?, +?, {m,n}?) match as little text as possible, stopping at the first valid match.

When using repetition in a regular expression, such as **a\*,** the pattern tries to consume as much input as possible. This [greedy behavior](https://youtu.be/fiar4QZZ7Xo?si=TP39KIEMOxOslXts) can cause issues when matching structures with balanced [delimiters](https://www.fynd.academy/blog/delimiters-in-python), like HTML tags. A common mistake when trying to match a single HTML tag is using **.\***, which consumes too much of the string.

For example, when applying **<.\*>** to an HTML snippet, the regex starts at the first **<**, then **.\*** greedily matches everything until the last possible **>**, leading to an incorrect match. The regex engine then backtracks, trying to find a valid **>** match, often resulting in an unintended selection.

To fix this, you can use non-greedy quantifiers like **\*?**, **+?**, **??**, or **{m,n}**, which match as little text as possible. This ensures that **>** is checked immediately after **<**, leading to the correct match. For example, using **<.\*?>** correctly captures only the first HTML tag rather than everything in between.

## 

## Examples

**^ (Caret) – Matches Start of a String**

import re

text = "hello world"

#text = " hello world"

pattern = r"^hello" # Match "hello" only if it's at the start

matches = re.findall(pattern, text)

print(matches) # Output: ['hello']

Output = [‘hello’]

**$ (Dollar) – Matches End of a String**

**import re**

**#text = "hello world"**

**text = " hello world!"**

**pattern = r"world$" # Match "world" only if it's at the end**

**matches = re.findall(pattern, text)**

**print(matches) # Output: ['world']**

Output = [ ]

**[] (Square Brackets) – Character Set**

text = "cat bat mat Hello"

pattern = r"[cb]at" # Matches 'cat' or 'bat'

#pattern = r"[cbm]at" # Matches 'cat' or 'bat' or mat

matches = re.findall(pattern, text)

print(matches) # Output: ['cat', 'bat']

Output = [‘cat’, ‘bat’]

**() (Parentheses) – Grouping & Capturing**

text = "John is the (Engineer) hello)"

pattern = r"\((.\*?)\)" # Captures text inside parentheses

matches = re.findall(pattern, text)

print(matches) # Output: ['Engineer']

Output = [‘Engineer’]

The pattern **\((.\*?)\)** looks for text enclosed within parentheses.

**\(** → The backslash \ is used to escape the opening parenthesis **(**, which is a special character in regex. This ensures that we are literally matching an opening parenthesis **(** in the text. **(.\*?)** → This part is the main capturing group, broken down as: **.\*** → Matches any character **(.)** appearing zero or more times **(\*)**. **?** (Lazy Match) → Makes the **\*** lazy, meaning it captures as little as possible instead of as much as possible.

**? (Question Mark) – Matches 0 or 1 Occurrence**

text = "helloo world"

pattern = r"lo?" # 'l' followed by 0 or 1 of 'o'

matches = re.findall(pattern, text)

print(matches) # Output: ['lo', 'l']

Output = ['l', 'lo', 'l']

**+ (Plus) – Matches 1 or More Occurrences, Matches the preceding character 1 or more times**.

text = "hellooo world"

pattern = r"lo+" # 'l' followed by at least one 'o'

matches = re.findall(pattern, text)

print(matches) # Output: ['looo']

Output = ['looo']

**\* (Asterisk) – Matches 0 or More Occurrences, Matches the preceding character many occurrences including zero**

text = "hellooo helo world"

pattern = r"lo\*" # 'l' followed by 'o' repeated 0 or more times

matches = re.findall(pattern, text)

print(matches) # Output: ['looo', 'lo']

Output = ['l', 'looo', 'lo', 'l']

**. (Dot) – Matches Any Character Except Newline (end of the line)**

import re

text = "cat bat mat"

#text = 'Monday Friday'

pattern = r"c.t" # 'c' followed by any character and 't'

#pattern =r'M.nday'

#pattern = r".t"

matches = re.findall(pattern, text)

print(matches)

Output = ['cat']

**| (Pipe) – OR Operator ,Matches either the left or right pattern.**

text = "apple orange banana"

pattern = r"apple|banana" # Matches "apple" OR "banana"

matches = re.findall(pattern, text)

print(matches)

Output = ['apple', 'banana']

**\ (Backslash) – Escape Special Characters, Escapes special characters to**

**match them literally.**

text = "Price: $100"

pattern = r"\$100" # Matches the dollar sign followed by 100

matches = re.findall(pattern, text)

print(matches)

Output = ['$100']

**{m,n } Examples**

import re

txt = "hello planet"

txt1 = "MOON PIE"

#Search for a sequence that starts with "he", followed excactly 2 (any) characters, and an "o":

x = re.findall("he.{2}o", txt)

#x = re.findall("M.{2}N", txt1)

print(x)

Output = [‘hello’]

import re

text = "Hellooo Heeelloo"

pattern = r"o{2}" # Matches exactly 3 occurrences of 'o'

#pattern = r"Hell.{3}"

matches = re.findall(pattern, text)

print(matches)

Output = ['oo', 'oo']

pattern = r"e{2,4}" # Matches 'e' between 2 and 4 times

matches = re.findall(pattern, text)

print(matches)

Output = [‘eee’]

text = "apple banana aaa abbb aaaa"

pattern = r"a{2,4}" # Matches 'a' repeated 2 to 4 times

matches = re.findall(pattern, text)

print(matches) # Output: ['aa', 'aaa', 'aaaa']

Output = [‘aaa’, ‘aaaa’]

**\d – Matches Any Digit (0-9)**

text = "My number is 12345"

pattern = r"\d+" # Matches one or more digits

matches = re.findall(pattern, text)

print(matches)

Output = [‘12345’]

**\D – Matches Any Non-Digit**

text = "My number is 12345"

pattern = r"\D+"

matches = re.findall(pattern, text)

print(matches)

Output = ['My number is ']

**\w – Matches Any Word Character (Alphanumeric & Underscore)**

text = "Hello\_World 123 how are you ! "

pattern = r"\w+"

matches = re.findall(pattern, text)

print(matches) # Output: ['Hello\_World', '123']

Output = ['Hello\_World', '123', 'how', 'are', 'you']

**\W – Matches Any Non-Word Character**

text = "Hello\_World 123 how are you ! "

pattern = r"\W+"

matches = re.findall(pattern, text)

print(matches)

Output = [' ', ' ', ' ', ' ', ' ! ']

**\s – Matches Any Whitespace (Space, Tab, Newline)**

text = "Hello World !"

pattern = r"\s+"

matches = re.findall(pattern, text)

print(matches)

Output = [' ', ' ']

**\S – Matches Any Non-Whitespace Character**

text = "Hello World !"

pattern = r"\S+"

matches = re.findall(pattern, text)

print(matches)

Output = ['Hello', 'World', '!']

**(?i) – Case Insensitive Matching**

text = "Python PYTHON python"

pattern = r"(?i)python"

matches = re.findall(pattern, text)

print(matches)

Output = ['Python', 'PYTHON', 'python']

**(?<=...) – Positive Lookbehind**

text = "Price: $100"

pattern = r"(?<=\$)\d+" # Matches numbers after $

matches = re.findall(pattern, text)

print(matches)

Output = ['100']

**(?=...) – Positive Lookahead**

text = "Python is awesome"

pattern = r"Python(?= is)"

matches = re.findall(pattern, text)

print(matches)

Output = ['Python']

[**Double Split**](https://youtu.be/GiQdXo2Bvgc?si=4wgFTfzMcavBBxva&t=135) **- Using split() twice**

line = "From rachel.mellark@georgiasouthernuniversity.edu on Tuesday April 5 10:23:54 2025"

words = line.split()

email = words[1]

print(words[1])

pieces = email.split('@')

print(pieces[1])

Output = rachel.mellark@georgiasouthernuniversity.edu

georgiasouthernuniversity.edu

Using [\b and \B](https://www.freecodecamp.org/news/what-does-b-in-regex-mean-word-boundary-and-non-word-boundary-metacharacters/#:~:text=%5Cw).-,How%20to%20Use%20the%20%E2%80%9CB%E2%80%9D%20Metacharacter%20in%20RegEx,-The%20%5Cb%20metacharacter):

**\b - Matches an empty string, but only at a word boundary**

import re

text = 'The war has taken its toll on the nation. Yet we must stay united in times of warfare.'

#this will find the exact matches for the word 'war' as a complete and independent word

match = re.findall(r'\bwar\b',text)

#notice how 'warfare' will not show when printed

print(match)

Output = [‘war’]

**\B - Matches the empty string, but not at the start or end of a word.**

import re

text = "singing is bringing everything into alignment."

# Find 'ing' that is not at the start of a word

matches = re.findall(r'\Bing', text)

print(matches)

Output = ['ing', 'ing', 'ing', 'ing', 'ing']

If you want to print the full words that contain ‘ing’ instead of just all instances on ‘ing’, follow the example below.

Notice this example uses \b, \w, and \B.

import re

text = "singing is bringing everything into alignment."

# Match full words containing 'ing' not at the start or end

matches = re.findall(r'\b\w\*\Bing\w\*\b', text)

print(matches)

Output = ['singing', 'bringing', 'everything']

More [{m,n}](https://youtu.be/P8Oq-kHJ5do?si=qIU_Ac7bgdmau1gL) examples:

{m,n}- matches the preceding element at least m times and at most n times.

text = "hellooo helo helooo helooooo"

match= re.findall(r"lo{2,4}", text)

print(match)

Output = ['looo', 'looo', 'loooo']

This regex lo{2,4} matches the letter "l" followed by 2 to 4 "o"s, so it finds patterns like "loo", "looo", or "loooo"—but not "lo" or "loooooo".

Another way of using {m,n} is by matching repeated digits in a string.

text = "Order 111 was placed on 2022-10-05 and order 22222 on 2023-01-01."

match = re.findall(r"\d{3,5}", text)

print(match)

Output = ['111', '2022', '22222', '2023']

The regex \d{3,5} matches any sequence of digits that’s between 3 and 5 digits long.

Another way of using {m,n} is to match specific number of words in a sentence.

text = "Order 111 was placed on 2022-10-05 and order 22222 on 2023-01-01."

match = re.findall(r"\b\w{3,6}\b", text)

print(match)

Output = ['Order', '111', 'was', 'placed', '2022', 'and', 'order', '22222', '2023']

## Reference and Additional Materials:

<https://docs.python.org/3/contents.html>

<https://www.dataquest.io/cheat-sheet/regular-expressions-cheat-sheet/>

<https://www.fynd.academy/blog/delimiters-in-python>

<https://www.pythontutorial.net/python-regex/>

<https://www.codecademy.com/resources/docs/python/regex>

<https://www.freecodecamp.org/>