**Mastering Python RegEx: A Deep Dive into Pattern Matching**

*Python RegEx Demystified: Deciphering the Art of Pattern Matching with Python’s re Module*

[[Nathan Rosidi](https://nathanrosidi.medium.com/?source=post_page-----2ae4fe6216fe--------------------------------)](https://nathanrosidi.medium.com/?source=post_page-----2ae4fe6216fe--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----2ae4fe6216fe--------------------------------)](https://towardsdatascience.com/?source=post_page-----2ae4fe6216fe--------------------------------)

[Nathan Rosidi](https://nathanrosidi.medium.com/?source=post_page-----2ae4fe6216fe--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fab636cbf3611&operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Fmastering-python-regex-a-deep-dive-into-pattern-matching-2ae4fe6216fe&user=Nathan+Rosidi&userId=ab636cbf3611&source=post_page-ab636cbf3611----2ae4fe6216fe---------------------post_header-----------)

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----2ae4fe6216fe--------------------------------)

·

16 min read

·

Jul 25, 2023

145

1

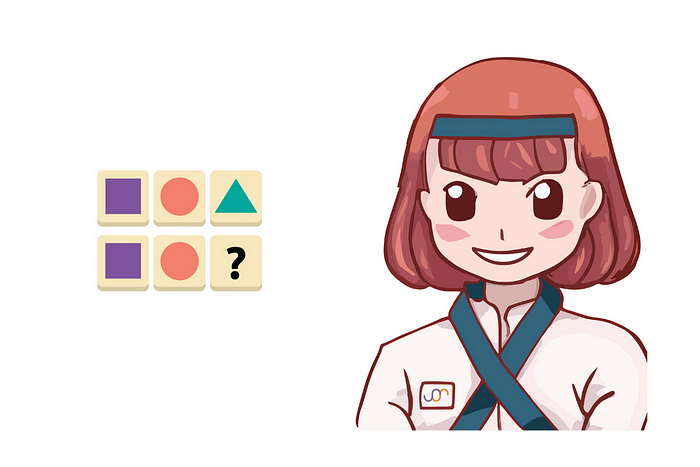


Image created by the author on [Canva](https://www.canva.com/)

**What is Python RegEx or Regular Expression?**

Regular expressions often shortened to regex, serve as a potent instrument for handling text. In essence, they consist of a series of characters that establish a pattern for searching. This pattern can be used for a wide range of string manipulations including matching patterns, replacing text, and dividing strings.

**History of Regular Expressions**



*Image Source:*[*http://serge.mehl.free.fr/chrono/Kleene.html*](http://serge.mehl.free.fr/chrono/Kleene.html)

Mathematician Stephen Cole Kleene first introduces regular expressions in the 1950s as a notation to describe regular sets or regular languages.

Today, regular expressions have become an essential skill for programmers, data scientists, and IT professionals.

**Importance and Use Cases of Python RegEx or Regular Expressions**

Before delving into how these regular expressions can be used, by using Python, let’s see the different range of its applications to motivate ourselves.

* **Data Validation**: Regular expressions can be very useful to validate different types of data. (e-mail addresses, phone numbers)
* **Web Scraping**: When scraping data through web pages, regular expressions can be used to parse HTML and isolate the necessary information.
* **Search and Replace**: Regular expressions are good at identifying strings that conform to a specific pattern and substituting them with alternatives. This capability is especially valuable in text editors, databases, and coding.
* **Syntax Highlighting:** A number of text editors use regular expressions to do syntax highlighting.
* **Natural Language Processing (NLP)**: Within NLP, regular expressions can be used for tasks such as tokenization, stemming, and an array of other text processing functions.
* **Log Analysis**: In dealing with log files, regular expressions are effective in extracting particular log entries or analyzing patterns over a period of time.

Now I hope, you are motivated enough!

Let’s get started with re module, which is all about Regular expressions.

**Getting Started with Python’s re Module**

Great, let’s get started with Python’s re module fundamentals. In the next sections, we will cover more advanced topics.

**Introduction to the re Module**

Python provides innate support for regular expressions via the re module.

This module is Python’s standard library, which means you don’t have to install it externally, it will come with every Python installation.

The re module contains various functions and classes to work with regular expressions. Some of the functions are used for matching text, some for splitting text, and others for replacing text.

It includes a wide range of functions an classes tailored for handling regular expressions. Amon these, certain functions are designated for text matching, remaining ones for text splitting or text replacements.

**Importing the re Module**

As we already mentioned, it came with installation, so no need to worry about installation.

That’s why, to start using regular expressions in Python, you need to import the re library first. You can do this by using the import statements as follows.

import re

After the library is imported, you can start its features like functions and classes, provided by the re module.

Let’s start with a simple example.

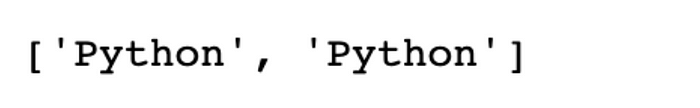
Let’s say you want to find all occurrences of the word “Python” in a string.

We can use the findall() function from the re module.

Here is the code.

import re   
# Sample text   
text = "Python is an amazing programming language. Python is widely used in various fields."   
# Find all occurrences of 'Python'   
matches = re.findall("Python", text)   
# Output the matches   
print(matches)

Here is the output.



There are many more functions in the re module that we can use to build more complex patterns.

But first, let’s see the common functions in the re Module.

**Common Functions in the re Module**

Before expressing to you the fundamentals of Python RegEx, let’s see the common functions first, to grasp the remaining concepts better. The re module includes many different functions. By using them, we can perform different operations.

In the following parts, we will discover some of them.

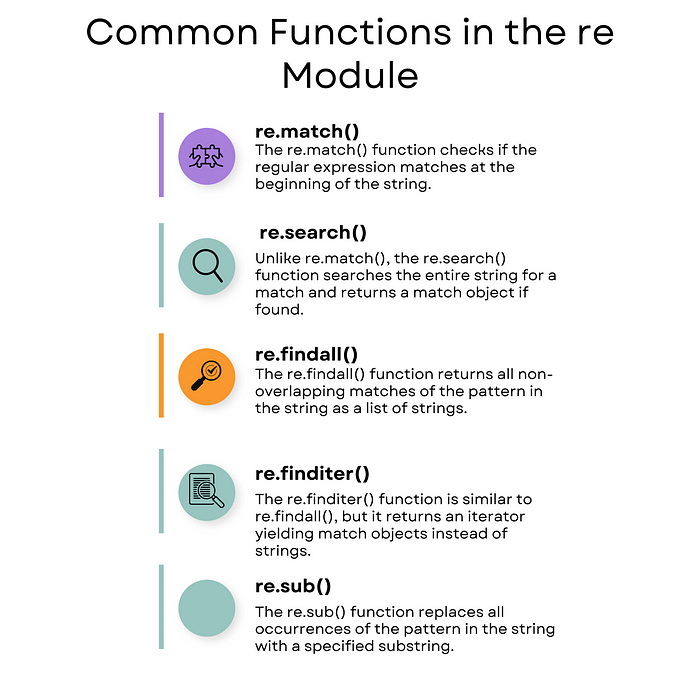


Image created by the author on [Canva](https://www.canva.com/)

**a. re.match() Function**

The re.match() catches whether the regular expression starts with the specific string or not.

If there is a match, the function returns a match object; if not, it returns none.

Next, we’ll use the re.match() function. Here we will check whether the string text starts with the word “Python” or not. Then we’ll print the result to the console.

Here is the code.

import re  
pattern = "Python"  
text = "Python is amazing."  
# Check if the text starts with 'Python'  
match = re.match(pattern, text)  
# Output the result  
if match:  
 print("Match found:", match.group())  
else:  
 print("No match found")

Here is the output.



The output shows that the pattern “Python” matches the beginning of the text.

**b. re.search() Function**

In contrast to re.match(), the re.search() function scans the entirety of the string in search of a match and yields a match object if one is discovered.

In the following code, we use the re.search() function to search for the word “amazing” anywhere in the string text. If the word is found, we print it; otherwise, we print “No match found”.

Here is the code.

pattern = "amazing"  
text = "Python is amazing."  
# Search for the pattern in the text  
match = re.search(pattern, text)  
# Output the result  
if match:  
 print("Match found:", match.group())  
else:  
 print("No match found")

Here is the output.



The output shows that our code catches amazing from the given text.

**c. re.findall() Function**

The re.findall() function is used to collect all the non-overlapping matches of a pattern in the string. And it returns these matches as a list of strings.

In the following example, we use the re.findall() function to find all “a” in the string. The matches are returned as a list, which then we print to the console.

Here is the code.

pattern = "a"  
text = "This is an example text."  
# Find all occurrences of 'a' in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches)

Here is the output.



The output represented all non-overlapping occurrences of the letter “a” found in our text.

**d. re.finditer() Function**

The re.finditer() function bears resemblance to re.findall(), however it returns an iterator, that yields match objects.

In the following code, re.finditer() function is used to find all occurrences of the letter “a” in the string text. It returns an iterator of match objects and we print the index and value of each match.

Here is the code.

pattern = "a"  
text = "This is an example text."  
# Find all occurrences of 'a' in the text  
matches = re.finditer(pattern, text)  
# Output the matches  
for match in matches:  
 print(f"Match found at index {match.start()}: {match.group()}")

Here is the output.



The output shows the index of the pattern “a” in the text.

**e. re.sub() Function**

The re.sub() function is used to do a replacement with one string to another.

Next, we’ll use the re.sub() function to replace “Python” with “Java”.

We then print the modified string.

Here is the code.

pattern = "Python"  
replacement = "Java"  
text = "I love Python. Python is amazing."  
# Replace 'Python' with 'Java'  
new\_text = re.sub(pattern, replacement, text)  
# Output the new text  
print(new\_text) # Output: "I love Java. Java is amazing."

Here is the output.



The output shows that we can successfully replace “Python” with “Java” from our text.

In the next section, we will discover into the basic patterns that can be used in regular expressions to match a variety of text patterns.

**Basic Patterns in Python Regular Expressions**

Let’s start with basic patterns.

Regular expressions are constructed through the combination of literal characters, meta-characters, and quantifiers. So, grasping these fundamental components is important for creating effective regular expressions.

Let’s start with literal characters.

**a. Literal Characters**

Literal characters are the simplest form of pattern matching in regular expressions.

They match themselves exactly and do not have a special meaning.

For example, the regular expression python will match the string python exactly.

import re   
pattern = "python"  
text = "I love programming in python!"   
# Find all occurrences of 'Python'   
matches = re.findall(pattern, text)   
# Output the matches   
print(matches)

Here is the output.



The output shows that our re.findall() function found all instances of the pattern “python”ç

**b. Meta-characters**

Meta-characters like “.”, “‘^”, “$”. These characters can be very important to manipulate strings. Let’s see.

**i. Dot (.)**

The dot . is like a Joker card. It can stand in for any single character except a newline.

In the code below, we’ll use a regular expression pattern “p.t”.

Here is the code.

import re   
pattern = "p.t"  
text = "pat, pet, p5t, but not pt."   
# Find all occurrences of 'Python'   
matches = re.findall(pattern, text)   
# Output the matches   
print(matches)

Here is the output.



The output shows that our code found all three character instances which start with “p” and ends with “t”.

**ii. Caret (^)**

The caret ^ is used to check if a string starts with a certain character.

Let’ see an example.

The following code checks whether the text starts with Hello( Match found : “match” ) or not ( No match found )

Here is the code.

import re  
pattern = "^Hello"  
text = "Hello, world!"  
# Use re.match() because it checks for a match only at the beginning of the string  
match = re.match(pattern, text)  
# Output the match  
if match:  
 print("Match found:", match.group())  
else:  
 print("No match found")

Here is the output.



The output shows that our code catches the hello pattern at the beginning of the text.

**iii. Dollar Sign ($)**

The dollar sign $ is used to check if a string ends with a certain character.

The following code checks whether the text ends with the world$ ( if so print “ Match found: “match) or not ( if so print “No match found” )

Here is the code.

import re  
pattern = "world$"  
text = "Hello, world"  
# Use re.search() to search the entire string  
match = re.search(pattern, text)  
# Output the match  
if match:  
 print("Match found:", match.group()) # Output: Match found: world  
else:  
 print("No match found")

Here is the output.



The output shows that re.search() function found the text that ends with the word “world”.

**c. Quantifiers**

Quantifiers are used to define how many times characters(or character) should appear in the pattern you are trying to match.

In this subsection, we will look at examples about the asterisk (\*), continue with the plus sign (+), and the question mark (?), and finish with curly braces ({}).

Let’s start with an asterisk.

**i. Asterisk (\*)**

The asterisk (\*) in a regular expression signifies that the previous character can appear zero or more times.

Let’s see the code. In the following code, we first define the pattern ( “py”), then we will use findall( ) function.

Here is the code.

import re  
  
pattern = "py\*"  
text = "p py pyy pyyy pyyyy"  
  
matches = re.findall(pattern, text)  
  
print(matches)

Here is the output.



The output shows all because asterisks allow for “y” to appear as zero or more times.

**ii. Plus (+)**

The plus + matches 1 or more repetitions of the previous character.

Here we again use findlall() function with the py pattern but this time we will use plus(+).

Here is the code.

import re  
  
pattern = "py+"  
text = "p py pyy pyyy pyyyy"  
  
matches = re.findall(pattern, text)  
  
print(matches) # Output: ['py', 'pyy', 'pyyy', 'pyyy']

Here is the output.



As we can see the output, plus requires at least one or more “y” characters after “p”.

**iii. Question Mark (?)**

The question mark ? matches 0 or 1 repetition of the previous character. It makes the previous character optional.

Here is the code.

import re  
  
pattern = "py?"  
  
text = "p py pyy pyyy pyyyy"  
  
matches = re.findall(pattern, text)  
  
print(matches) # Output: ['p', 'py', 'p', 'p', 'p']

Here is the output.



In the output, you can see that it only matches “p” and “py”, since question mark allows to appear “y” one time or zero times.

**iv. Curly Braces ({})**

Curly braces {} allow you to match a specific number of repetitions.

import re  
  
pattern = "py{2,3}"  
text = "py, pyy, pyyy, pyyyy"  
  
matches = re.findall(pattern, text)  
  
print(matches) # Output: ['pyy', 'pyyy', 'pyy']

Here is the output.



In this example, the pattern matches “pyy” and “pyyy” but not “py” or “pyyyy” because we specified that we want to match exactly 2 or 3 “y” characters after “p”.

**Special Characters in Python Regular Expressions**

Special can be used to build more complex patterns.



Image created by the author on [Canva](https://www.canva.com/)

**a. Character Classes**

Let’s see character classes first.

In the following examples, we will see 3 of them.

Let’s start with \d, \D.

**i. \d, \D**

The “\d” is used to find numbers (from 0 to 9), on the contrary, “\D” is used to find elements that are not numbers.

In the following code, “\d” scans through the text string and retrieve numbers from the text.

import re  
pattern = "\d"  
text = "My phone number is 123-456-7890."  
# Find all digits in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches)

Here is the output.



The output shows that we found all digits (0–9) in the text.

**ii. \s, \S**

The “\s” can be used to find whitespace characters, on the opposite “\S can be used to find anything that is not whitespace.

In the below, the regular expression “\s” identifies all spaces and tabs in the given text.

Here is the code.

import re  
pattern = "\s"  
text = "This is a text with spaces and\ttabs."  
# Find all whitespace characters in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches) # Output: [' ', ' ', ' ', ' ', ' ', ' ', '\t']

Here is the output.



We can see from the outputs that we can identify all the whitespace.

**iii. \w, \W**

The “\w” can be used to find words. (letters, numbers, and underscore characters)”\W” is the opposite of that.

In the code below, “\w” retrieves all letters and numbers from the text.

Here is the code.

import re  
pattern = "\w"  
text = "This is an example with words and numbers 123!"  
# Find all word characters in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches)

Here is the output.



**b. Predefined Character Classes**

Predefined character classes offer shortcuts for common classes. For example, “\d” is a predefined character class that represents digits.

In this case, the “\d” pattern extracts all numerical digits from the given text.

import re  
pattern = "\d"  
text = "The year is 2023."  
# Find all digits in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches)

Here is the output.



The output shows that our code has found all instances of predefined character class “\d” (representing all digits) in the text.

**c. Custom Character Classes**

Custom character classes allow you to define your own set of characters using square brackets [].

In the example below, the custom character class “[aeiou]” is used to find all vowel letters in the text.

Here is the code.

import re  
pattern = "[aeiou]"  
text = "This is an example text."  
# Find all vowels in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches)

Here is the output.



The output shows all instances of vowels in the text as we defined it.

We also can use “-” to define the range of characters.

Here is the code.

pattern = "[A-Z]"  
text = "This is an Example Text With Uppercase Letters."  
# Find all uppercase letters in the text  
matches = re.findall(pattern, text)  
# Output the matches  
print(matches)

Here is the output.



Here we can the output consists of the uppercase letters in the text.

**Compiling Python Regular Expressions**

When you use the same regular expression multiple times in a script, it is time-saving to compile it into a pattern object first. This saves a lot of time because the regular expression doesn’t need to be parsed again with each use.

**a. The compile() Method**

The re.compile() method can be used to compile a regular expression pattern into a pattern object.

Once we have this pattern object, we can call its methods (matching text, searching, and other operations.)

Here is the code.

import re  
# Compile the regular expression pattern  
pattern = re.compile(r'\d+') # Matches one or more digits  
# Use the pattern object to search for matches  
text = "There are 3 apples and 4 oranges."  
matches = pattern.findall(text)  
# Output the matches  
print(matches)

Here is the output.



The output shows digits.

**b. Benefits of Compiling Regular Expressions**

Here are some benefits of using regular expressions;

* **Performance**: It is faster, especially if the regular expressions will be used again and again.
* **Reusability**: Once compiled, the same pattern object can be reused multiple times within different parts of the code.
* **Readability**: Using a pattern object can make your code cleaner, especially if you are using complex regular expressions.

Here is a simple example of compiled regular expressions:

import re  
# Compile the regular expression pattern  
pattern = re.compile(r'\d+') # Matches one or more digits  
# Use the pattern object to search for matches in different texts  
text1 = "There are 3 apples."  
text2 = "I have 15 dollars and 30 cents."  
# Find matches in text1  
matches1 = pattern.findall(text1)  
# Find matches in text2  
matches2 = pattern.findall(text2)  
# Output the matches  
print(matches1)

Here is the output.



Now let’s check the second text.

Here is the code.

print(matches2)

Here is the output.



Our example above is rather a simple one for you to grasp the importance of reusability, performance, and readability, especially when our pattern plan to use repeatedly.

**Practical Example: Extracting Phone Numbers**

In this section, let’s test what we discover together by writing a Python script to extract phone numbers from text.

This one is a common use of regular expressions, especially in the data-cleaning process.

**a. Defining the Regular Expression Pattern**

Phone numbers can be in different formats, especially in different countries, so you can adjust these numbers according to yours, for this example, let’s consider the format XXX-XXX-XXXX, where X is a digit.

The following code defines a pattern that matches the format above and complies with this pattern into a regular expression.

Let’s see the code.

import re  
# Define the regular expression pattern for phone numbers  
phone\_number\_pattern = re.compile(r'\d{3}-\d{3}-\d{4}')

**b. Using the findall() Method**

In this example, we will use findall() method to extract phone numbers that matched our pattern.

The following code uses a regular expression pattern to find and extract all

import re  
# Define the regular expression pattern for phone numbers  
phone\_number\_pattern = re.compile(r'\d{3}-\d{3}-\d{4}')  
  
  
# Sample text with phone numbers  
text = """  
John Doe: 123-456-7890  
Jane Doe: 234-567-8901  
Office: 555-555-5555  
"""  
# Find all phone numbers in the text  
phone\_numbers = phone\_number\_pattern.findall(text)

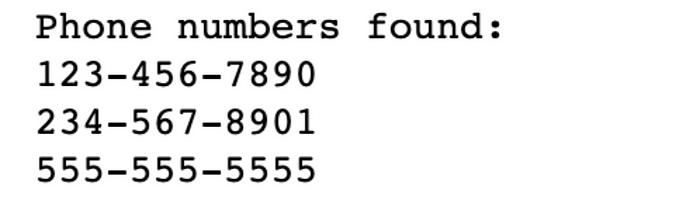
**c. Printing the Results**

Finally, let’s print the extracted phone numbers to the console.

Here is the code.

# Output the phone numbers  
print("Phone numbers found:")  
for phone\_number in phone\_numbers:  
 print(phone\_number)

Here is the output.

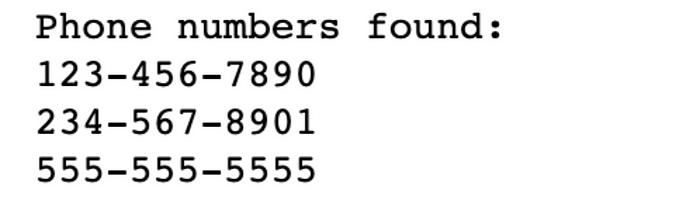


**d. Full Example Code**

Here is the full Python script that combines all the steps above:

import re  
# Define the regular expression pattern for phone numbers  
phone\_number\_pattern = re.compile(r'\d{3}-\d{3}-\d{4}')  
# Sample text with phone numbers  
text = """  
John Doe: 123-456-7890  
Jane Doe: 234-567-8901  
Office: 555-555-5555  
"""  
# Find all phone numbers in the text  
phone\_numbers = phone\_number\_pattern.findall(text)  
# Output the phone numbers  
print("Phone numbers found:")  
for phone\_number in phone\_numbers:  
 print(phone\_number)

Here is the output.



**Best Practices**

As you continue to work with regular expressions, here are a few best practices to keep in mind:

* **Keep it Simple**: Simplicity is the key. It is generally advised to use a simpler pattern because regular expressions can be complicated instantly.
* **Comment Your Patterns**: When developing regular expressions for your project, don’t forget to include comments inside your notes, as we told it can be complicated, but once you did this when you turn back, your code will be reusable.
* **Test Thoroughly**: Test your code over and over again, because regular expressions can sometimes produce unexpected results due to their complex nature, that is why testing it rigorously will ensure your work will work as intended.
* **Use Raw Strings**: When you’re working with text in Python, sometimes you use special characters that have a different meaning than just the character itself (like backslash \ is or \n for a new line ). To avoid this confusion, Python allows you to use what’s called a “raw string”. You make a string “raw” by putting the letter “r” right before the first quote of the string. When you do this, Python understands that backslashes in that string should be treated just like normal characters and not as special ones.

**Conclusion**

In this guide, we explored the realm of Python RegEx or Regular Expressions. We started with common functions and fundamentals and go through more advanced concepts and practical examples. But remember doing real-life projects, that will count as an example for your career to deepen this understanding of your mind. Just by doing so, you’ll develop knowledge and save yourself from googling whenever you work on Python regular expressions.

Check out this [comprehensive guide to advanced Python concepts](https://www.stratascratch.com/blog/a-comprehensive-guide-to-advanced-python-concepts/?utm_source=blog&utm_medium=click&utm_campaign=kdn+python+regex) to get an overview of such concepts.

I hope you also gained valuable information about Python RegEx by reading this article too.

Thanks for reading!