

Python

Mastering Python Regular Expressions

Pattern Matching Made Simple

May 21, 2025



Source Code

1. Introduction to Regular Expressions

Regular expressions (regex) are powerful tools for pattern matching and text manipulation in Python. They provide a concise and flexible way to:

- Search and match specific patterns in text
- Validate string formats (emails, phone numbers, etc.)
- Extract information from structured text
- Perform complex string replacements

Python's `re` module provides comprehensive support for regular expressions, making it an essential tool for text processing tasks.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast



www.asanchezyali.com

Regex Quick Reference

Pattern	Meaning
.	Any character except newline
*	Zero or more repetitions
+	One or more repetitions
?	Zero or one repetition
[a-z]	Any lowercase letter
[A-Z]	Any uppercase letter
[0-9]	Any digit
[abc]	a, b, or c
\d	Digit (0-9)
\w	Word character (a-z, A-Z, 0-9, _)
\s	Whitespace
^	Start of string
\$	End of string
	Alternation (or)
()	Grouping
\	Escape special character


2. Basic Pattern Matching

Let's start with fundamental regex concepts and pattern matching:



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast


 www.asanchezyali.com

```
1 """
2 Basic Pattern Matching in Python
3
4 This file demonstrates fundamental regex patterns and matching.
5
6 Quick Reference:
7 .      Any character except newline
8 *      Zero or more repetitions
9 +      One or more repetitions
10 ?     Zero or one repetition
11 [a-z]  Any lowercase letter
12 [A-Z]  Any uppercase letter
13 [0-9]  Any digit
14 \d     Digit (0-9)
15 \w     Word character (a-z, A-Z, 0-9, _)
16 \s     Whitespace
17 ^     Start of string
18 $     End of string
19 |     Alternation (or)
20 ()     Grouping
21
22 Try your regex online: regex101.com, pythex.org, regexr.com
23 """
24 import re
25
26 def display_match(pattern: str, text: str, description: str) -> None:
27     """Helper function to display regex matches."""
28     matches = re.findall(pattern, text)
29     print(f"\n{description}:")
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast


 www.asanchezyali.com

```
30     print(f"Pattern: {pattern}")
31     print(f"Text: {text}")
32     print(f"Matches: {matches}")
33
34 # Visual Example 1: Simple match
35 text1 = "The quick brown fox jumps over the lazy dog"
36 pattern1 = r"fox"
37 match = re.search(pattern1, text1)
38 print(f"Simple match - found '{pattern1}' at position:
      {match.start()}-{match.end()}")
39
40 # Visual Example 2: Case-insensitive matching
41 pattern2 = r"FOX"
42 match = re.search(pattern2, text1, re.IGNORECASE)
43 print(f"\nCase-insensitive match found: {match.group()}")
44
45 # Visual Example 3: Word boundaries
46 text2 = "firefox is not a fox but firefox contains fox"
47 pattern3 = r"\bfox\b"
48 matches = re.findall(pattern3, text2)
49 print(f"\nWords that are exactly 'fox': {matches}")
50
51 # Visual Example 4: Multiple patterns using |
52 text3 = "The cat and the dog play with another cat"
53 pattern4 = r"cat|dog"
54 matches = re.findall(pattern4, text3)
55 print(f"\nFinding 'cat' or 'dog': {matches}")
56
57 # Visual Example 5: Greedy vs. non-greedy matching
58 text4 = "<tag>content</tag>"
59 greedy_pattern = r"<.*>"
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
60 non_greedy_pattern = r"<.*?>"
61 print(f"\nGreedy match: {re.findall(greedy_pattern, text4)}")
62 print(f"Non-greedy match: {re.findall(non_greedy_pattern, text4)}")
63
64 # Visual Example 6: ASCII word matching (replaces Unicode example)
65 text5 = "Cafe naive resume"
66 pattern5 = r"\w+"
67 print(f"\nASCII word matches (default): {re.findall(pattern5, text5)}")
68 print(f"ASCII word matches (with re.UNICODE): {re.findall(pattern5, text5,
    re.UNICODE)}")
```

Key concepts demonstrated:

- The `r` prefix creates a raw string
- `re.search()` finds the first match in text
- `re.IGNORECASE` flag makes the search case-insensitive
- `re.findall()` returns all matches
- Word boundaries with `\b`

3. Visual Examples

Example 1: Simple Match

Pattern: fox

String: The quick brown fox jumps over the lazy dog

Match: fox



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

Example 2: Digit Extraction

Pattern: `\d+`

String: Order numbers: 123, 456, 789

Match: 123, 456, 789

4. Common Pattern Validation


Regular expressions are frequently used to validate common text formats. Here are practical examples:

```
1 """
2 Common Pattern Validation Examples
3
4 This file demonstrates how to validate common text patterns using regex.
5
6 Try your regex online: regex101.com, pythex.org, regexr.com
7 """
8 import re
9
10 def validate_pattern(pattern: str, text: str) -> str:
11     """Helper function to validate if text matches pattern."""
12     try:
13         return "[VALID]" if re.match(pattern, text) else "[INVALID]"
14     except re.error as e:
15         print(f"Regex error: {e}")
16         return "[ERROR]"
17
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com


PYTHON | REGEX

```
18 # Email validation
19 # Pattern: [alphanumeric and special]@[domain].[TLD]
20 email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
21 emails = [
22     "user@example.com",
23     "invalid.email@com",
24     "name.surname+tag@domain.co.uk",
25     "@invalid.com",
26     "spaces are@not.allowed.com"
27 ]
28
29 print("Email Validation:")
30 for email in emails:
31     print(f"{email}: {validate_pattern(email_pattern, email)}")
32
33 # Phone number validation
34 # Pattern: optional +, optional 1, 9-15 digits
35 description = "Phone number: optional +, optional 1, 9-15 digits"
36 phone_pattern = r"^\+?1?\d{9,15}$"
37 phones = [
38     "+1234567890",
39     "123-456-7890",
40     "12345",
41     "+442012345678"
42 ]
43
44 print("\nPhone Number Validation:")
45 for phone in phones:
46     print(f"{phone}: {validate_pattern(phone_pattern, phone)}")
47
48 # URL validation
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast


 www.asanchezyali.com


```
49 # Pattern: http(s)://... (basic)
50 url_pattern = r"https?:\/\/(?:[\w.-]+\.)+[\w.-]+(?:\/[\w.\/?%&=-]*)?$"
51 urls = [
52     "https://www.example.com",
53     "http://sub.domain.co.uk/path?param=value",
54     "not-a-url.com",
55     "https://api.site.com/v1/data.json"
56 ]
57
58 print("\nURL Validation:")
59 for url in urls:
60     print(f"{url}: {validate_pattern(url_pattern, url)}")
61
62 # Date validation (YYYY-MM-DD)
63 # Pattern: 4 digits-2 digits-2 digits, with month/day checks
64 date_pattern = r"^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])$"
65 dates = [
66     "2025-12-31",
67     "2025-13-01",
68     "2025-04-31",
69     "25-12-31"
70 ]
71
72 print("\nDate Validation (YYYY-MM-DD):")
73 for date in dates:
74     print(f"{date}: {validate_pattern(date_pattern, date)}")
75
76 # Multiline and DOTALL flag demonstration
77 multiline_text = """First line
78 Second line
79 Third line"""
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
80 pattern_multiline = r"^Second"
81 print("\nMultiline flag demo:")
82 print(re.findall(pattern_multiline, multiline_text, re.MULTILINE))
83
84 pattern_dotall = r"First.*Third"
85 print("DOTALL flag demo:")
86 print(re.findall(pattern_dotall, multiline_text, re.DOTALL))
87
88 # Common extraction recipes
89 sample_text = "Contact: user@example.com, visit https://site.com, call
    +1234567890, date 2025-12-31."
90 print("\nExtract emails:",
    re.findall(r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}", sample_text))
91 print("Extract URLs:", re.findall(r"https?://[\w.-]+(?:/[\w./?%&=-]*)?",
    sample_text))
92 print("Extract dates:", re.findall(r"\d{4}-\d{2}-\d{2}", sample_text))
93 print("Extract phone numbers:", re.findall(r"\+?1?\d{9,15}", sample_text))
```


Important pattern components:

- `^` and `$` anchor the pattern to start/end
- `[a-zA-Z0-9]` matches any alphanumeric character
- `+` matches one or more occurrences
- `*` matches zero or more occurrences
- `?` makes a character optional
- `\d` matches any digit
- `\w` matches word characters



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

5. Advanced Features

Regular expressions support advanced features such as:

- **Named capture groups:** `(?P<name>pattern)`
- **Non-capturing groups:** `(?:pattern)`
- **Positive lookahead:** `(?=pattern)`
- **Negative lookahead:** `(?!pattern)`
- **Positive lookbehind:** `(?<=pattern)`
- **Negative lookbehind:** `(?<!=pattern)`
- **Greedy vs. non-greedy quantifiers:** `*` vs. `*?`
- **Backreferences:** `\1`, `\2`, etc.
- **Pattern substitution:** `re.sub()`

Visual Example: Non-greedy Matching

Pattern: `<.*?>`


String: `<tag>content</tag>`

Match: `<tag>`, `</tag>`



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

6. Testing Your Regex

You can experiment with regular expressions interactively using online tools such as:

- regex101.com
- pythex.org
- regexr.com

7. Common Regex Recipes

- **Extract emails:** `[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}`
- **Extract URLs:** `https?://[\w.-]+(?:/[\w./?%&=-]*)?`
- **Extract dates (YYYY-MM-DD):** `\d{4}-\d{2}-\d{2}`
- **Extract phone numbers:** `\+?1?\d{9,15}`


8. Error Handling and Useful Flags

- Use `try/except` to catch `re.error` exceptions in Python.
- Useful flags:
 - `re.IGNORECASE` (case-insensitive)
 - `re.MULTILINE` (^ and \$ match at line breaks)



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- `re.DOTALL` (dot matches newline)
- `re.UNICODE` (Unicode matching)
- `re.VERBOSE` (allow comments and whitespace in pattern)

9. Best Practices

When working with regular expressions in Python:

- Use raw strings (`r"pattern"`) to avoid escaping backslashes
- Start with simple patterns and gradually add complexity
- Test patterns with a variety of inputs
- Use `re.VERBOSE` for complex patterns with comments
- Consider performance with large texts
- Document complex patterns
- Use existing patterns for common formats

10. Common Pitfalls


Be aware of these common regex pitfalls:

- Catastrophic backtracking with nested quantifiers
- Greedy vs. non-greedy matching



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- Unicode handling in patterns
- Over-complicated patterns
- Not escaping special characters
- Incorrect character class usage

11. Performance Tips

Optimize your regex usage:

- Compile patterns you use repeatedly with `re.compile()`
- Use non-capturing groups `(?:...)` when possible
- Avoid unnecessary backtracking
- Be specific with character classes
- Consider alternative solutions for complex patterns
- Profile pattern matching on large datasets


12. References

- Python Documentation. (2025). *re - Regular expression operations*. [Link](#)
- Goyvaerts, J., & Levithan, S. (2023). *Regular Expressions Cookbook*. O'Reilly Media.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- This article was edited and written in collaboration with AI. If you find any inconsistencies or have suggestions for improvement, please don't hesitate to open an issue in our [GitHub](#) repository or reach out directly.

13. Explore More Python Concepts

Enjoyed This Content?

Don't miss my other Python articles:

Python Namespaces: A Deep Dive into Name Resolution

Learn how Python organizes and manages variable names, scopes, and namespaces.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast



www.asanchezyali.com



Feedback

Found this helpful?

Save, comment and share

May 21, 2025