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.

2. Basic Pattern Matching

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

```python
"""
Basic Pattern Matching in Python

This file demonstrates fundamental regex patterns and matching.
"""
import re

def display_match(pattern: str, text: str, description: str) -> None:
    """Helper function to display regex matches."""
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
    matches = re.findall(pattern, text)
    print(f"\n{description}:")
    print(f"Pattern: {pattern}")
    print(f"Text: {text}")
    print(f"Matches: {matches}")

# Basic pattern matching
text1 = "The quick brown fox jumps over the lazy dog"
pattern1 = r"fox"
match = re.search(pattern1, text1)
print(f"Simple match - found '{pattern1}' at position:
    {match.start()}-{match.end()}")

# Case-insensitive matching
pattern2 = r"FOX"
match = re.search(pattern2, text1, re.IGNORECASE)
print(f"\nCase-insensitive match found: {match.group()}")

# Word boundaries
text2 = "firefox is not a fox but firefox contains fox"
display_match(r"\bfox\b", text2, "Words that are exactly 'fox'")

# Multiple patterns using |
text3 = "The cat and the dog play with another cat"
display_match(r"cat|dog", text3, "Finding 'cat' or 'dog'")
```

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

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- `re.findall()` returns all matches
- Word boundaries with `\b`

3. Common Pattern Validation

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

```python
"""
Common Pattern Validation Examples

This file demonstrates how to validate common text patterns using regex.
"""
import re

def validate_pattern(pattern: str, text: str) -> str:
    """Helper function to validate if text matches pattern."""
    return "[VALID]" if re.match(pattern, text) else "[INVALID]"

# Email validation
email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
emails = [
    "user@example.com",
    "invalid.email@com",
    "name.surname+tag@domain.co.uk",
    "@invalid.com",
    "spaces are@not.allowed.com"
]
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **REGEX**

```python

print("Email Validation:")
for email in emails:
    print(f"{email}: {validate_pattern(email_pattern, email)}")

# Phone number validation
phone_pattern = r"^\+?1?\d{9,15}$"
phones = [
    "+1234567890",
    "123-456-7890",
    "12345",
    "+442012345678"
]

print("\nPhone Number Validation:")
for phone in phones:
    print(f"{phone}: {validate_pattern(phone_pattern, phone)}")

# URL validation
url_pattern = r"https?://(?:[\w.-]+\.)+[\w.-]+(?:/[\w./?%&=-]*)?$"
urls = [
    "https://www.example.com",
    "http://sub.domain.co.uk/path?param=value",
    "not-a-url.com",
    "https://api.site.com/v1/data.json"
]

print("\nURL Validation:")
for url in urls:
    print(f"{url}: {validate_pattern(url_pattern, url)}")

```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
# Date validation (YYYY-MM-DD)
date_pattern = r"^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])$"
dates = [
    "2025-12-31",
    "2025-13-01",
    "2025-04-31",
    "25-12-31"
]

print("\nDate Validation (YYYY-MM-DD):")
for date in dates:
    print(f"{date}: {validate_pattern(date_pattern, date)}")
```

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

4. Advanced Features

Regular expressions become more powerful with advanced features:

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **REGEX**

```python
"""
Advanced Regular Expression Features

This file demonstrates advanced regex features including groups,
lookahead/lookbehind assertions, and substitutions.
"""

import re
from typing import List, Tuple


def parse_log_entry(log_line: str) -> dict:
    """Parse a log entry using named capture groups."""
    pattern = r"""
        ^                           # Start of line
        (?P<timestamp>\d{4}-\d{2}-\d{2}\s\d{2}:\d{2})\s+  # Date and time
        \[(?P<level>INFO|WARN|ERROR)\]\s+                 # Log level
        \[(?P<module>[\w.-]+)\]\s+                        # Module name
        (?P<message>.+)$                                  # Log message
    """
    match = re.match(pattern, log_line, re.VERBOSE)
    return match.groupdict() if match else {}


# Example with named groups
log_line = "2025-05-10 14:30 [ERROR] [user.auth] Failed login attempt"
parsed = parse_log_entry(log_line)
print("Parsed Log Entry:")
for key, value in parsed.items():
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
    print(f"{key}: {value}")


def extract_methods(code: str) -> List[Tuple[str, str]]:
    """Extract method names and their return types using lookahead."""
    pattern = r"def\s+(\w+)\s*\([^)]*\)\s*->\s*([^:]+):"
    return re.findall(pattern, code)


# Example with method extraction
python_code = """
def calculate_total(items: List[float]) -> float:
    pass

def process_data(data: dict) -> List[str]:
    pass
"""

print("\nExtracted Methods:")
for method, return_type in extract_methods(python_code):
    print(f"Method: {method}, Returns: {return_type}")


# Example with substitution and backreferences
def clean_text(text: str) -> str:
    """Clean text by removing repeated words."""
    pattern = r"\b(\w+)\s+\1\b"  # Pattern for repeated words
    return re.sub(pattern, r"\1", text)


text = "The the quick quick brown fox"
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

PYTHON | **REGEX**

```python
cleaned = clean_text(text)
print(f"\nOriginal text: {text}")
print(f"Cleaned text: {cleaned}")


# Example with positive/negative lookahead
def find_prices(text: str) -> None:
    """Find prices with different currencies using capturing groups."""
    # Matches numbers (with optional decimals), optional spaces, and currency
    pattern = r"(\d+(?:\.\d+)?)(?:\s*)(USD|EUR|GBP)"
    print("\nPrices found:")
    for match in re.finditer(pattern, text):
        amount = match.group(1)
        currency = match.group(2)
        print(f"Amount: {amount}, Currency: {currency}")


text = "The price is 100 USD, 200 EUR, and 150 GBP"
find_prices(text)
```

Advanced concepts covered:

- Named capture groups with `(?P<name>pattern)`
- `re.VERBOSE` flag for readable patterns
- Positive/negative lookahead with `(?=...)` and `(?!...)`
- Backreferences with `\1`, `\2`, etc.
- Pattern substitution with `re.sub()`

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

5. 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

6. Common Pitfalls

Be aware of these common regex pitfalls:

- Catastrophic backtracking with nested quantifiers
- Greedy vs. non-greedy matching
- Unicode handling in patterns
- Over-complicated patterns
- Not escaping special characters
- Incorrect character class usage

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

7. 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

8. References

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

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

9. Explore More Python Concepts

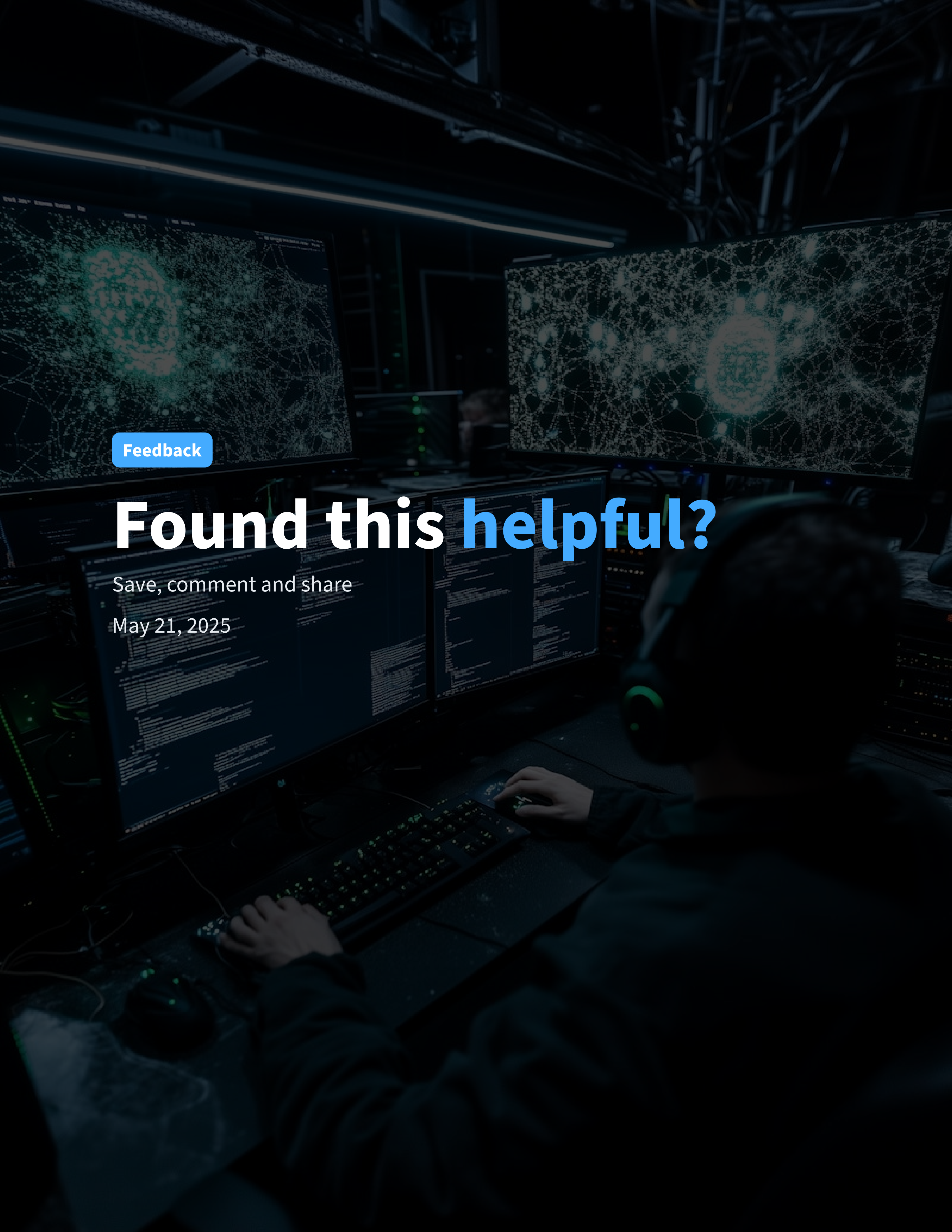
Enjoyed This Content?

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