Natural Language Processing

An Introduction

Abdelbacet Mhamdi

2025-01-23

ISET Bizerte

Outline

1. Introduction to Regular Expressions (Regex)

1.1 What are Regular Expressions?

Regular expressions are powerful patterns used to match, search, and manipulate text strings. They provide a standardized way to describe search patterns in text, making them an essential tool in programming, text processing, and data validation.

1.2.1 Pattern Matching

A regex pattern is a sequence of characters that defines a search pattern. These patterns can be:

- Literal characters that match themselves;
- Special characters (metacharacters) with special meanings;
- Combinations of both.

1.2.2 Basic Metacharacters

Metacharacter	Description	Example
.	Matches any character except newline	a.c matches “abc”, “a1c”, “a@c”
^	Matches start of string	^Hello matches “Hello World”
$	Matches end of string	world$ matches “Hello world”
*	Matches 0 or more occurrences	ab*c matches “ac”, “abc”, “abbc”
+	Matches 1 or more occurrences	ab+c matches “abc”, “abbc” but not “ac”
?	Matches 0 or 1 occurrence	ab?c matches “ac” and “abc”
\	Escapes special characters	\. matches literal dot

1.3 Common Use Cases

1. **Search Operations**
   - Advanced find/replace operations;
   - Pattern matching in large text files;
   - Content filtering.
2. **Text Processing**
   - Finding patterns in text;
   - Replacing specific text patterns;
   - Extracting information;
   - Parsing log files.
3. **Data Validation**
   - Email addresses;
   - Phone numbers;
   - Postal codes;
   - Passwords;
   - URLs.

1.4.1 Character Classes

```python
# Character class examples
pattern = r'[aeiou]'  # Matches any vowel
pattern = r'[0-9]'    # Matches any digit
pattern = r'[^0-9]'   # Matches any non-digit
```

1.4.2 Quantifiers and Groups

```python
# Quantifiers
pattern = r'\d{3}'      # Exactly 3 digits
pattern = r'\d{2,4}'    # Between 2 and 4 digits
pattern = r'\d{2,}'     # 2 or more digits

# Groups
pattern = r'(\w+)\s+\1'  # Matches repeated words
```

1.4.3 Common Regex Functions in Python

Python

```python
import re

text = "The price is $19.99"

# Different matching functions
re.search(r'\$\d+\.\d+', text)   # Finds first match
re.findall(r'\$\d+\.\d+', text)  # Finds all matches
re.sub(r'\$(\d+\.\d+)', r'\1', text)  # Substitution

# Splitting text
re.split(r'\s+', text)  # Split on whitespace
```

1.5.1 Basic Pattern Matching

Python

```python
import re
# Simple pattern matching
text = "The quick brown fox jumps over the lazy dog"
pattern = r"fox"
# Search for pattern
match = re.search(pattern, text)
if match:
    print(f"Found '{pattern}' at position: {match.start()}-{match.end()}")
# Find all occurrences
words = re.findall(r"\w+", text)
print(f"All words: {words}")
```

1.5.2 Email Validation Example

Python

```python
def is_valid_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))
# Test cases
emails = [
    "user@example.com", # ✔
    "invalid.email@com", # ✗
    "user.name@bizerte.r-iset.tn", # ✔
    "@invalid.com" # ✗
]
for email in emails:
    print(f"{email} → {'Valid' if is_valid_email(email) else 'Invalid'}")
```

1.5.3 Phone Number Formatting

Python

```python
def format_phone_number(phone):
    # Remove all non-digit characters
    digits = re.sub(r'\D', '', phone)

    # Format as (XXX) XXX-XXXX
    if len(digits) == 10:
        pattern = r'(\d{3})(\d{3})(\d{4})'
        formatted = re.sub(pattern, r'(\1) \2-\3', digits)
        return formatted

    return "Invalid phone number"
```

1.5 Python Implementation

Python

```python
# Test cases
numbers = [
    "1234567890",
    "123-456-7890",
    "(123) 456-7890",
    "12345"
]

for number in numbers:
    print(f"{number} → {format_phone_number(number)}")
```

1.6 Best Practices

1. **Use Raw Strings**
   - Always prefix regex patterns with `r` to avoid escape character issues

```python
pattern = r'\d+'  # Better than '\d+'
```

2. **Compile Frequently Used Patterns**

```python
email_pattern = re.compile(r'^[\w\.-]+@[\w\.-]+\.\w+$')
# Use multiple times
email_pattern.match(email1)
email_pattern.match(email2)
```

3. **Be Specific**
   - Make patterns as specific as possible to avoid false matches;
   - Use start (^) and end ($) anchors when matching whole strings.

4. **Test Thoroughly**
   - Test with both valid and invalid inputs
   - Include edge cases in your tests

```python
def test_pattern(pattern, test_cases):
    regex = re.compile(pattern)
    for test, expected in test_cases:
        result = bool(regex.match(test))
        print(f"'{test}': {'✓' if result == expected else '✗'}")
```

1. **Greedy vs. Non-Greedy Matching**

```python
# Greedy (default)
re.findall(r'<.*>', '<tag>text</tag>')  # ['<tag>text</tag>']

# Non-greedy: Add (lazy) `?`
re.findall(r'<.*?>', '<tag>text</tag>')  # ['<tag>', '</tag>']
```

2. **Performance Considerations**
   - Avoid excessive backtracking *(recursion)*;
   - Be careful with nested quantifiers;
   - Use more specific patterns when possible.

1. **Basic Pattern Matching**

```python
# Write a pattern to match dates in format DD/MM/YYYY
date_pattern = r'\d{2}/\d{2}/\d{4}'
```

2. **Data Extraction**

```python
# Extract all email addresses from text
text = "Contact us at support@example.com or sales@example.com"
emails = re.findall(r'[\w\.-]+@[\w\.-]+\.\w+', text)
```

3. **Password Validation**

```python
def is_strong_password(password):
    # At least 8 chars, 1 upper, 1 lower, 1 digit, 1 special
    pattern = r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$' # Positive Lookahead
    return bool(re.match(pattern, password))
```

1.8 Applications

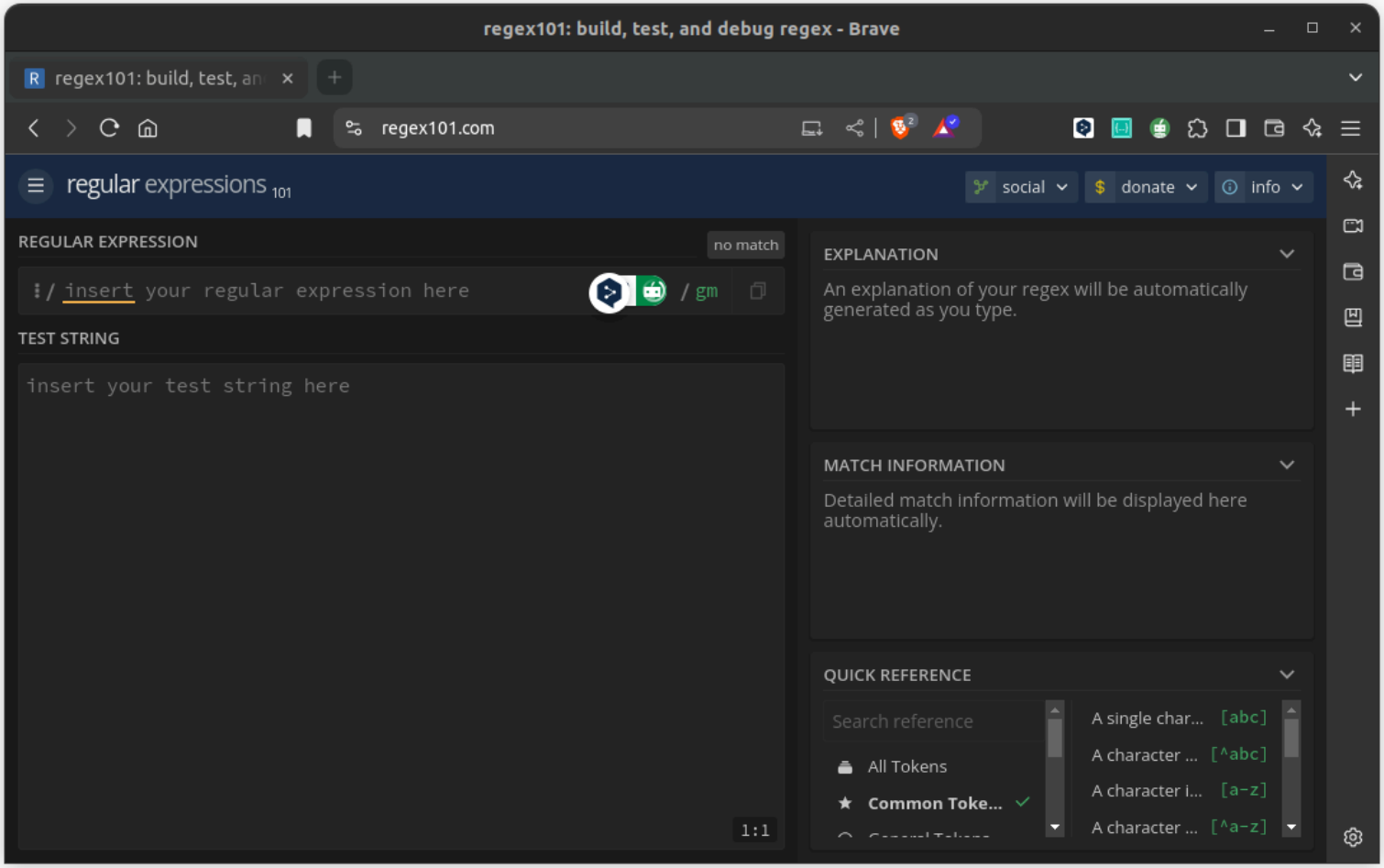

Figure 1: Build, test and debug regex patterns.

Thank you for your attention!