WRITE BETTER PATTERNS. SOLVE COMPLEX PROBLEMS.
WORK CONFIDENTLY ACROSS ANY PLATFORM.

MASTERING REGULAR EXPRESSIONS

A COMPLETE GUIDE FOR DEVELOPERS ACROSS LANGUAGES AND PLATFORMS

FROM ZERO TO EXPERT
Build a strong foundation and advance with confidence.

CLEAR EXPLANATIONS AND WALKTHROUGHS
Understand how regex works, not just the syntax.

PRACTICAL EXAMPLES YOU CAN USE
Real-world scenarios and copy-and-adapt solutions.

DEBUG, OPTIMIZE, AND TEST
Practical techniques to write better, faster, and safer regex.

WRITE REGEX YOU CAN TRUST
Maintainable, readable, and future-proof patterns.

EXAMPLES AND SOLUTIONS ACROSS
9 PROGRAMMING LANGUAGES

PYTHON | PERL | RUBY | GO | RUST | C | C++ | JAVA | C#

ERIC M. LIANG

Mastering Regular Expressions

A Complete Guide for Developers Across Languages and Platforms

Steve Publications

This book is available at https://leanpub.com/masteringregularexpressions

This version was published on 2026-08-21

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Complete Guide for Developers Across Languages and Platforms . . . **1**

Introduction: Why Regular Expressions Matter . . . **2**
What This Book Covers . . . 2
Languages and Engines Covered . . . 3
How to Use This Book . . . 4
A Note on Notation . . . 5

Chapter 1: What Are Regular Expressions? . . . **6**
The Power of Pattern Matching . . . 6
A Brief History: From Automata Theory to grep . . . 7
What Regex Can and Cannot Do . . . 8
How This Book Is Organized: Your Learning Roadmap . . . 10

Chapter 2: Your First Regular Expressions . . . **12**
Literal Characters and Exact Matches . . . 12
Writing Your First Pattern . . . 17
The Anatomy of a Regex Operation . . . 19
Testing Patterns Interactively . . . 20

Chapter 3: Metacharacters and Escaping . . . **23**
The Dot Metacharacter . . . 23
Special Characters That Need Escaping . . . 23
Escape Sequences for Common Characters . . . 23
When Escaping Behavior Differs Across Engines . . . 23

Chapter 4: Character Classes and Ranges . . . **24**
Defining Character Sets with Square Brackets . . . 24
Ranges and Ordering Within Classes . . . 24
Negated Character Classes . . . 24
Shorthand Character Classes . . . 24

POSIX Character Classes 24

Chapter 5: Quantifiers and Repetition 25
The Asterisk, Plus, and Question Mark 25
Exact and Range Quantifiers 25
Greedy Matching Explained 25
Common Quantifier Mistakes 25

Chapter 6: Anchors and Boundaries 26
Start and End of String Anchors 26
Word Boundaries 26
Line Anchors and Multiline Mode 26
Zero-Width Assertions vs Consuming Characters 26

Chapter 7: Alternation and Grouping 27
The Pipe Operator for Alternation 27
Capturing Groups with Parentheses 27
Non-Capturing Groups 27
Prioritizing Alternatives 27

Chapter 8: Backreferences and Named Groups 28
Numeric Backreferences 28
Named Capture Groups 28
Backreferences in Replacement Strings 28
Engine Support Variations 28

Chapter 9: Lookahead and Lookbehind Assertions 29
Positive Lookahead 29
Negative Lookahead 29
Positive Lookbehind 29
Negative Lookbehind 29
Fixed-Length vs Variable-Length Lookbehind 29

Chapter 10: Greedy, Lazy, and Possessive Quantifiers 30
Understanding Greedy Behavior Step by Step 30
Lazy (Reluctant) Quantifiers 30
Possessive Quantifiers and Their Performance Benefits 30
Choosing the Right Quantifier Mode 30

Chapter 11: Atomic Groups and Advanced Assertions 31
Atomic Groups and Backtracking Control 31

Conditional Patterns 31
Recursive and Balanced Matching 31
Engine-Specific Features 31

Chapter 12: Unicode and International Text 32
Unicode Code Points vs Bytes 32
Unicode Property Escapes 32
Case Insensitivity Across Languages 32
Grapheme Clusters and Combining Characters 32

Chapter 13: Matching, Searching, Extracting, Replacing 33
Full Match vs Search vs Find All 33
Extracting Captured Groups 33
Splitting Strings on Patterns 33
Replacement with Backreferences 33

Chapter 14: Real-World Pattern Recipes 34
Email Address Validation 34
URL and URI Matching 34
Date and Time Patterns 34
File Paths and Filenames 34
Log Line Parsing 34

Chapter 15: Source Code and Structured Text Processing 35
Tokenizing Source Code 35
Parsing Configuration Files 35
Working with Markup Languages 35
When to Use a Parser Instead 35

Chapter 16: Debugging and Testing Regex 36
Interactive Testing Tools 36
Writing Unit Tests for Regex 36
Step-by-Step Debugging Techniques 36
Common Pitfalls and How to Spot Them 36

Chapter 17: Performance and Security 37
Understanding Backtracking Cost 37
Catastrophic Backtracking Explained 37
Regular Expression Denial of Service (ReDoS) 37
Writing Linear-Time Patterns with RE2 37

Chapter 18: Maintainability, Portability, and Best Practices **38**
Readable Regex: Comments and Formatting 38
Porting Patterns Across Engines 38
When Not to Use Regular Expressions 38
Documentation and Team Standards 38

Chapter 19: Regex Engines and Flavors Compared **39**
NFA vs DFA vs Hybrid Engines 39
Engine Feature Comparison Table 39
Unicode Handling Differences 39
Performance Characteristics 39

Chapter 20: Language-Specific API Guides **40**
Python (re and regex modules) 40
Perl (m//, s///, qr//) 40
Ruby (Regexp class) 40
Go (regexp package) 40
Rust (regex crate) 40
C (POSIX regex.h and PCRE2) 40
C++ (std::regex) 41
Java (java.util.regex) 41
C# (.NET System.Text.RegularExpressions) 41

Chapter 21: Putting It All Together **42**
The Regex Decision Framework 42
Building Your Pattern Library 42
Continuing Your Learning Journey 42
Final Thoughts 42

Glossary of Terms **43**

Syntax and Metacharacter Cheat Sheet **44**
Core Metacharacters 44
Quantifiers 44
Shorthand Character Classes 44
Common Escape Sequences 44

Common Patterns Reference **45**
Validation Patterns 45
Extraction Patterns 45

Transformation Patterns 45

Troubleshooting Guide 46
Pattern does not match expected input 46
Pattern matches too much (greedy problem) 46
Pattern is very slow or hangs 46
Pattern works in one language but not another 46
Unicode-related issues 46

Further Reading and Bibliography 47
Books 47
Papers and Technical Articles 47
Official Documentation 47
Standards 47
Online Tools 47

References 48

A Complete Guide for Developers Across Languages and Platforms

Regular expressions are one of the most powerful tools in a developer's toolkit, yet they remain among the most misunderstood. This book takes you from zero knowledge to expert-level mastery through clear explanations, step-by-step walkthroughs, and practical examples across nine programming languages including Python, Perl, Ruby, Go, Rust, C, C++, Java, and C#. Every concept is explained not just as syntax to memorize but as a mechanism you can reason about, debug, optimize, and port safely across different regex engines.

Introduction: Why Regular Expressions Matter

Imagine you are handed a log file with two million lines. Somewhere in those lines are authentication failures from a specific IP range between 02:00 and 04:00 on Tuesday nights. Your manager needs a report by morning. You could write a parser that tokenizes each line, checks timestamps, extracts IPs, and applies business logic. Or you could run one command with a pattern that does it all in seconds.

That pattern is a regular expression.

Regular expressions are concise descriptions of text patterns. They let you search for strings that contain certain words, extract email addresses from documents, validate user input, parse configuration files, clean messy data, and transform text systematically. Every major programming language includes regex support because the problems they solve appear constantly in real software.

Yet regular expressions have a reputation for being cryptic, fragile, and impossible to read once written. Much of that reputation is deserved. A pattern copied from the internet without understanding can fail silently on edge cases, run forever on certain inputs, or behave differently depending on which language you paste it into. The goal of this book is to replace that uncertainty with competence.

What This Book Covers

This book is organized as a progressive learning path from absolute beginner through expert-level material. You do not need any prior experience with regular expressions to start reading. If you know basic programming concepts like strings, loops, and functions, you have enough background.

The first part introduces the fundamental ideas: what a regex pattern is, how literal characters match, and how special metacharacters give regex its

power. You will write your first patterns immediately and see exactly how an engine processes them step by step.

The second and third parts build those foundations into practical skill. You will learn character classes that match sets of letters or digits, quantifiers that control repetition, anchors that constrain where matches can occur, grouping that organizes complex patterns, and backreferences that let a pattern refer to text it has already matched.

The fourth part covers advanced features that separate competent regex users from experts: lookahead and lookbehind assertions that check context without consuming characters, atomic groups and possessive quantifiers that control backtracking behavior for performance and correctness, conditional patterns, recursive matching, and Unicode-aware operations for international text.

The fifth part focuses on real-world application. You will work through production-quality patterns for email addresses, URLs, dates, file paths, log parsing, source code processing, and structured-text extraction. Each pattern is explained thoroughly so you can adapt it to your own needs rather than treating it as a black box.

The sixth and seventh parts address engineering concerns that matter when regex becomes part of a real system: debugging strategies, testing methodology, performance optimization, security risks including Regular Expression Denial of Service attacks, maintainability practices, portability across engines, and guidance on when regex is the right tool versus when you should reach for something else.

Throughout the book, every important concept follows a consistent structure. First, the objective is stated clearly: what problem this feature solves. Next, the pattern is broken down piece by piece so you understand exactly what each part does. Then you see representative input data, the complete regex syntax, runnable code in multiple languages, expected output, and a step-by-step walkthrough of how the engine processes the pattern and produces the result. Finally, practical notes cover edge cases, common mistakes, performance considerations, and compatibility differences across engines.

Languages and Engines Covered

This book provides equivalent examples wherever reasonably possible in nine programming languages:

- Python (using the standard library re module)
- Perl (using built-in m//, s///, qr// operators)
- Ruby (using Regexp with the Onigmo engine)
- Go (using the regexp package based on RE2)
- Rust (using the regex crate)
- C (using POSIX regex.h and PCRE2 library examples)
- C++ (using std::regex from the standard library)
- Java (using java.util.regex.Pattern and Matcher)
- C# (using .NET System.Text.RegularExpressions)

These languages represent a wide spectrum of regex engine designs. Some use powerful backtracking engines that support every feature but can run slowly or hang on bad patterns. Others prioritize guaranteed linear-time performance by omitting certain features. Understanding the differences is critical because a pattern that works perfectly in one language may fail, behave differently, or never terminate in another.

When a feature is unsupported in a particular language, this book says so explicitly rather than providing misleading workarounds. Portability is not just about syntax; it is about understanding which engines support which features and designing patterns accordingly.

How to Use This Book

Read the chapters in order if you are new to regular expressions. Each chapter assumes only what earlier chapters have taught, and the progression from simple literal matching through advanced engine-specific features mirrors how expertise actually develops.

If you already know regex basics, you can skip ahead to the sections most relevant to your needs: lookarounds and assertions for advanced pattern design, performance and security chapters if you work with user-supplied

input or large datasets, language-specific API guides when starting a new project, and real-world pattern recipes for practical solutions.

The examples are designed to be runnable as-is. Copy them into your editor, run them, modify them, and experiment. The best way to learn regex is to see patterns work on real text and then change something to observe what happens.

A Note on Notation

Throughout this book, regular expression patterns appear in monospace font like this: `^[a-z]+@[a-z0-9.-]+\.[a-z]{2,}$`. Code examples use language-appropriate syntax highlighting where possible. When discussing regex engines or flavors, the full name is used initially (for example "PCRE2" for Perl Compatible Regular Expressions version 2) and then abbreviated consistently thereafter.

Inline citations appear as numbers in square brackets like [1] immediately after the claims they support. The complete reference list with links to all sources appears at the end of the book.

Now let us begin with the foundations: what regular expressions actually are, where they came from, and why they work the way they do.

Chapter 1: What Are Regular Expressions?

The Power of Pattern Matching

Every day, software processes text. User input arrives in forms, log files accumulate entries, configuration files define settings, APIs exchange JSON and XML, source code repositories store millions of lines of structured text. Most of this text is not completely free-form; it follows patterns, conventions, and structures that can be described precisely.

A regular expression is a compact language for describing those patterns. Instead of writing code that checks each character individually with nested conditions, you write a single pattern string that encodes the entire rule set. The regex engine then does the work of finding matches, extracting components, validating format, or replacing text.

Consider a practical example: validating that a user has entered something resembling an email address. Without regex, you might write code like this in Python:

```
def is_email(s):
    if '@' not in s:
        return False
    local, domain = s.rsplit('@', 1)
    if '.' not in domain:
        return False
    if len(local) == 0 or len(domain) == 0:
        return False
    # ... more checks for allowed characters, length limits, etc.
    return True
```

With regex, the same validation becomes a single pattern and one function call:

```
import re
pattern = re.compile(r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$')
is_valid = bool(pattern.match(user_input))
```

The regex version is more concise and often easier to modify. Want to allow Unicode characters in the local part? Change one character class. Want to enforce a minimum domain length? Adjust a quantifier. The pattern itself documents the rule.

But regex power comes with responsibility. A poorly designed pattern can be slow, incorrect on edge cases, or vulnerable to abuse. Understanding how regex works under the hood is not academic: it directly affects correctness, performance, and security in production systems.

A Brief History: From Automata Theory to grep

Regular expressions originated in theoretical computer science, far from any practical text-processing tool. In 1951, mathematician Stephen Cole Kleene introduced "regular events" as a mathematical notation for describing sets of strings [1]. His work built on earlier research by Warren McCulloch and Walter Pitts on neural networks and laid the foundation for automata theory: the study of abstract machines that recognize patterns in input sequences.

Kleene's notation described what we now call regular languages: sets of strings that can be recognized by finite automata, simple state machines with no memory beyond their current state. The operations were straightforward: concatenation (putting patterns together), alternation (choosing between alternatives), and repetition (repeating a pattern zero or more times). These three operations remain the core of regex syntax today.

The leap from theory to practice happened in 1968 when Ken Thompson, one of the creators of Unix, implemented Kleene's notation into a text editor called QED running on the Compatible Time-Sharing System at Bell Labs [2]. Thompson needed an efficient way to search for patterns in text files, and he solved it by compiling regular expressions into nondeterministic finite automata (NFAs) and then simulating those automata using just-in-time compilation to IBM 7094 machine code. This was one of the earliest examples of JIT compilation in computing history.

Thompson's regex implementation moved from QED to the Unix editor ed, which appeared in First Edition Unix in 1971. From ed came grep, a standalone

utility that searched files for lines matching a given regular expression. The name "grep" comes from the ed command g/re/p: globally search for a regular expression and print matching lines [3]. Grep first appeared in Fourth Edition Unix in 1973 and became one of the most influential tools in computing.

Throughout the 1970s, regex spread through Unix culture via grep, sed (stream editor), and awk (pattern-directed language). The POSIX standard eventually formalized two flavors: Basic Regular Expressions (BRE) and Extended Regular Expressions (ERE), which differ mainly in which characters are treated as metacharacters versus literals [4].

The modern era of regex began with Perl. In the late 1980s, Larry Wall designed Perl's regex engine to be both powerful and convenient for text processing tasks. Perl added features that went beyond formal regular languages: backreferences (referring to previously matched text), non-greedy quantifiers, lookahead and lookbehind assertions, and named capture groups. These extensions made regex practical for parsing complex text structures but also introduced complexity in how patterns are evaluated.

Philip Hazel started writing PCRE (Perl Compatible Regular Expressions) in 1997 as a C library that implemented Perl-style regex syntax independently of the Perl interpreter [5]. PCRE became enormously influential: it is used by PHP, Apache, Nginx, R, and countless other projects. PCRE2, its successor with a revised API, continues active development today.

Meanwhile, Google developed RE2 as an alternative approach. Russ Cox's 2007 paper "Regular Expression Matching Can Be Simple And Fast" demonstrated that the backtracking algorithms used by Perl, Python, Java, and others could be exponentially slow on certain inputs, while Thompson's original NFA algorithm maintained linear time [6]. RE2 implements a principled design: it guarantees linear-time matching by using a combination of DFA and Thompson NFA techniques, at the cost of omitting features like backreferences that require exponential-time algorithms in the worst case.

This history matters because it explains why regex engines differ so dramatically today. Some prioritize feature richness (PCRE, Perl, .NET), others prioritize guaranteed performance (RE2, Rust's regex crate), and some try to balance both. The pattern syntax you write may look similar across languages, but the engine underneath can behave very differently.

What Regex Can and Cannot Do

Understanding the formal limits of regular expressions helps explain why modern engines include features that go "beyond" classical regular languages, and where those extensions encounter their own limits.

Formally, a regular language is any set of strings recognizable by a finite automaton. Regular expressions in the original Kleene sense describe exactly these languages. They can express patterns like:

- Strings consisting entirely of digits
- Sequences of alternating letters and numbers
- Text containing a specific word anywhere within it
- Patterns with fixed or bounded repetition

What regular languages cannot do is count arbitrarily. You cannot write a classical regular expression that matches strings with an equal number of opening and closing parentheses, because doing so requires unbounded memory to track the nesting depth. Similarly, you cannot match palindromes of arbitrary length or validate properly nested HTML tags using only formal regular expressions.

Modern regex engines sidestep this limitation through extensions. Backreferences let a pattern refer back to text it has already matched. The pattern `^(\w+)\s+\1$` matches lines where the same word appears twice separated by whitespace: "hello hello" matches, but "hello world" does not. The `\1` is a backreference to whatever was captured by the first group `(\w+)`. This capability allows matching of non-regular languages, which means the worst-case time complexity can become exponential rather than linear.

Even with these extensions, regex has practical limits:

- Deeply nested structures like HTML, XML, or JSON are better parsed with dedicated parsers that build parse trees. Regex can extract simple patterns from markup but will struggle with arbitrary nesting.
- Context-sensitive validation (for example, "this field must match the format defined in that other field") is usually clearer as explicit code rather than regex.
- Very long patterns become difficult to read, debug, and maintain regardless of engine capabilities.

The key insight is this: regex excels at pattern matching within a single string or line where the structure can be described by local rules about character sequences. When you need global understanding of document structure, hierarchical relationships, or semantic meaning, other tools are more appropriate. Regex is not a replacement for parsers; it is a complementary tool that shines in its own domain.

How This Book Is Organized: Your Learning Roadmap

This book is structured as a journey from zero to mastery, divided into seven parts that build progressively on each other. You should read them in order if you are new to regex, but experienced users can navigate directly to relevant sections.

Part I: Foundations establishes the basic concepts. You will learn what regular expressions are historically and theoretically, write your first patterns, understand how literal characters and metacharacters work, and grasp the fundamental idea that a pattern describes a set of possible strings rather than a single specific string.

Part II: Core Building Blocks covers the essential features you will use in almost every regex: character classes for matching sets of characters, quantifiers for controlling repetition, anchors for constraining match positions, alternation for choosing between options, and grouping for organizing complex patterns. By the end of this part, you will be able to write functional regex patterns for common tasks.

Part III: Intermediate Patterns introduces features that let you express more sophisticated constraints: backreferences for matching repeated text, named groups for readable capture references, and modifiers that change how patterns behave. These tools transform basic patterns into production-ready solutions.

Part IV: Advanced Features covers the techniques used by expert regex practitioners: lookahead and lookbehind assertions for context-aware matching without consuming characters, atomic groups and possessive quantifiers for controlling backtracking behavior, conditional patterns, recursive matching for nested structures, and Unicode-aware operations for international text. These features are powerful but require careful understanding to use correctly.

Part V: Practical Applications focuses on real-world usage. You will see complete, tested patterns for email validation, URL matching, date formats, file paths, log parsing, source code processing, and structured-text extraction. Each pattern is explained thoroughly so you understand not just what it matches but why it works that way and how to adapt it.

Part VI: Engineering Concerns addresses the practical challenges of using regex in production systems: debugging strategies for understanding why a pattern does or does not match, testing methodology for verifying correctness across edge cases, performance optimization techniques, security risks including Regular Expression Denial of Service attacks, maintainability practices for keeping patterns readable, and guidance on when regex is appropriate versus when other approaches are better.

Part VII: Language and Engine Reference provides comprehensive reference material: comparison tables showing which features are supported by which engines, language-specific API guides for all nine languages covered in this book, syntax cheat sheets, a glossary of terms, and troubleshooting references. This part is designed to be consulted as needed rather than read linearly.

The learning path assumes no prior regex knowledge but moves quickly once fundamentals are established. If you find yourself confused at any point, revisit the earlier chapters: regex concepts build on each other, and gaps in foundational understanding make advanced topics much harder to grasp.

Now let us write your first regular expression.

Chapter 2: Your First Regular Expressions

Literal Characters and Exact Matches

The simplest possible regular expression is one that matches exactly what you type. If you write the pattern `cat`, it matches the three characters c, a, t appearing consecutively in that order. Nothing more, nothing less.

This literal matching behavior is the foundation of everything else in regex. Every pattern ultimately consists of specifications for which characters to match at which positions. Metacharacters and special syntax layers on top of this basic idea, but literal characters are always present.

Consider searching through a text file for occurrences of the word "error". The pattern `error` will find every instance where those five letters appear consecutively:

- In the string "An error occurred", it matches starting at position 3
- In "errors were logged", it matches starting at position 0 (the first five characters)
- In "no problems here", it finds no match

The pattern does not care about word boundaries, case sensitivity, or context. It simply looks for the exact sequence of characters specified. This simplicity is both a strength and something to be aware of: `error` will also match inside words like "erroneous" or "terrorist". We will learn how to constrain matching later, but understanding this basic behavior first is essential.

Let us see literal matching in action across multiple languages. The objective here is straightforward: given some input text, find whether the exact pattern appears anywhere within it.

Input data:

```
The quick brown fox jumps over the lazy dog.
```

Pattern: `fox`

Here is how you would search for this literal pattern in each language:

Python:

```python
import re

text = "The quick brown fox jumps over the lazy dog."
pattern = re.compile("fox")
match = pattern.search(text)

if match:
    print(f"Found '{match.group()}' at position {match.start()}")
else:
    print("No match found")
```

Output:

```
Found 'fox' at position 16
```

Perl:

```perl
my $text = "The quick brown fox jumps over the lazy dog.";

if ($text =~ /fox/) {
    print "Match found: '$&'\n";
} else {
    print "No match found\n";
}
```

Output:

```
Match found: 'fox'
```

Ruby:

```ruby
text = "The quick brown fox jumps over the lazy dog."

if text =~ /fox/
  puts "Match found at position #{$~.begin(0)}"
else
  puts "No match found"
end
```

Output:

```
Match found at position 16
```

Go:

```go
package main

import (
    "fmt"
    "regexp"
)

func main() {
    text := "The quick brown fox jumps over the lazy dog."
    pattern := regexp.MustCompile("fox")
    match := pattern.FindStringIndex(text)

    if match != nil {
        fmt.Printf("Found '%s' at position %d\n", pattern.FindString(text),
        ↪  match[0])
    } else {
        fmt.Println("No match found")
    }
}
```

Output:

```
Found 'fox' at position 16
```

Rust:

```rust
use regex::Regex;

fn main() {
    let text = "The quick brown fox jumps over the lazy dog.";
    let pattern = Regex::new("fox").unwrap();
    if let Some(m) = pattern.find(text) {
        println!("Found '{}' at position {}", m.as_str(), m.start());
    } else {
        println!("No match found");
    }
}
```

Output:

```
Found 'fox' at position 16
```

C (using POSIX regex.h):

```c
#include <stdio.h>
#include <regex.h>
#include <string.h>

int main() {
    const char *text = "The quick brown fox jumps over the lazy dog.";
    regex_t regex;
    regmatch_t match;

    if (regcomp(&regex, "fox", REG_EXTENDED) != 0) {
        fprintf(stderr, "Failed to compile regex\n");
        return 1;
    }

    if (regexec(&regex, text, 1, &match, 0) == 0) {
        printf("Found '%.3s' at position %td\n",
               text + match.rm_so, match.rm_so);
    } else {
        printf("No match found\n");
    }

    regfree(&regex);
    return 0;
}
```

Output:

```
Found 'fox' at position 16
```

C++ (using std::regex):

```cpp
#include <iostream>
#include <regex>
#include <string>

int main() {
    std::string text = "The quick brown fox jumps over the lazy dog.";
    std::regex pattern("fox");
    std::smatch match;

    if (std::regex_search(text, match, pattern)) {
        std::cout << "Found '" << match[0] << "' at position "
                  << match.position(0) << std::endl;
    } else {
        std::cout << "No match found" << std::endl;
    }

    return 0;
}
```

Output:

```
Found 'fox' at position 16
```

Java:

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LiteralMatch {
    public static void main(String[] args) {
        String text = "The quick brown fox jumps over the lazy dog.";
        Pattern pattern = Pattern.compile("fox");
        Matcher matcher = pattern.matcher(text);

        if (matcher.find()) {
            System.out.println("Found '" + matcher.group() +
                "' at position " + matcher.start());
        } else {
            System.out.println("No match found");
        }
    }
}
```

Output:

```
Found 'fox' at position 16
```

C# (.NET):

```
using System;
using System.Text.RegularExpressions;

class LiteralMatch {
    static void Main() {
        string text = "The quick brown fox jumps over the lazy dog.";
        Regex pattern = new Regex("fox");
        Match match = pattern.Match(text);

        if (match.Success) {
            Console.WriteLine($"Found '{match.Value}' at position
            ↪  {match.Index}");
        } else {
            Console.WriteLine("No match found");
        }
    }
}
```

Output:

```
Found 'fox' at position 16
```

Notice that all nine languages produce the same result for this simple literal pattern. The API differs significantly: Python uses `re.compile` and `.search()`, Perl uses the `=~` operator with `/pattern/`, Go uses `regexp.MustCompile` and `.FindStringIndex()`, Java uses separate Pattern and Matcher classes, and so on. But the regex syntax itself is identical because a literal string pattern is universal across all engines.

Writing Your First Pattern

Now let us write something slightly more interesting: a pattern that matches a phone number in the format 123-456-7890. This introduces two concepts: matching specific characters like the hyphen, and understanding that each character in the pattern corresponds to one character position in the input.

Objective: Match a ten-digit phone number in XXX-XXX-XXXX format where X is any digit from 0 to 9.

For now, let us match the literal digits of a specific number: `555-123-4567`. The pattern is simply those characters typed out:

Pattern: `555-123-4567`

Input data:

```
Contact us at 555-123-4567 for support.
Our fax number is 555-987-6543.
```

This pattern will match only in the first line, at the exact phone number specified. It will not match the second line because the digits are different. This is correct behavior for literal matching but clearly not useful if we want to match any phone number in this format.

To make the pattern flexible, we need a way to say "match any digit" rather than matching one specific digit. That capability comes from character classes and shorthand escapes, which we will cover in detail in Chapter 4. For now, let us preview the solution:

Pattern: `\d{3}-\d{3}-\d{4}`

Here `\d` means "any digit" and `{3}` means "exactly three times". We will unpack this syntax thoroughly later. The point for now is to see how a pattern evolves from matching one specific string to matching an entire class of strings that share a structure.

Engine walkthrough: When the engine processes `\d{3}-\d{3}-\d{4}` against "Contact us at 555-123-4567 for support", it works as follows:

1. The engine starts scanning from position 0 of the input
2. At positions 0 through 12 ("Contact us at "), `\d` fails to match letters and spaces, so the engine advances
3. At position 13, it encounters "5", which is a digit, so `\d` matches
4. The `{3}` quantifier requires three digits total, so the engine checks positions 13, 14, 15: all are digits (5-5-5), satisfying `\d{3}`
5. The next part of the pattern is a literal hyphen `-`, and position 16 contains `-`, so it matches
6. Positions 17 through 19 contain "123", three more digits, satisfying the second `\d{3}`

7. Position 20 has another hyphen, matching the literal `-`
8. Positions 21 through 24 contain "4567", four digits, satisfying `\d{4}`
9. The entire pattern has matched successfully from position 13 to position 24 (inclusive of start, exclusive of end in most APIs)

This step-by-step scanning behavior is fundamental to how regex engines work. They try to match the pattern starting at each position in the input until they find a successful match or exhaust all positions. We will explore this process in much greater depth when we discuss quantifiers and backtracking.

The Anatomy of a Regex Operation

Every regex operation, regardless of language or engine, involves three core components:

1. **The pattern:** A string describing what to match (for example, `\d{3}-\d{3}-\d{4}`)
2. **The input text:** The string being searched (for example, "Call 555-123-4567 now")
3. **The operation type:** What you want to do with the match

Common operation types include:

- **Search:** Find the first occurrence of the pattern anywhere in the input. Returns whether a match exists and where it is located.
- **Match (full):** Check if the entire input matches the pattern from start to end. More restrictive than search.
- **Find all:** Locate every non-overlapping occurrence of the pattern in the input.
- **Split:** Divide the input into pieces using matches of the pattern as delimiters.
- **Replace:** Find matches and substitute them with replacement text.

Different languages expose these operations through different APIs, but they all perform the same fundamental tasks. Let us look at how search versus full match differs, because confusing these two is a very common mistake.

Python example showing the difference:

```python
import re

text = "abc123def"

# Search: finds pattern anywhere in the string
search_result = re.search(r"\d+", text)
print("Search result:", search_result.group() if search_result else None)
# Output: Search result: 123

# Full match: requires entire string to match pattern
full_result = re.fullmatch(r"\d+", text)
print("Full match result:", full_result.group() if full_result else None)
# Output: Full match result: None

# Full match succeeds only when the whole string is digits
full_result2 = re.fullmatch(r"\d+", "123")
print("Full match on '123':", full_result2.group() if full_result2 else None)
# Output: Full match on '123': 123
```

In Perl, the distinction appears as different operators or anchoring:

```perl
my $text = "abc123def";

# Search: finds digits anywhere
if ($text =~ /\d+/) {
    print "Search found: $&\n";
}  # Output: Search found: 123

# Full match using anchors ^ and $
if ($text =~ /^\d+$/) {
    print "Full match succeeded\n";
} else {
    print "Full match failed\n";
}  # Output: Full match failed
```

In Java, the distinction is between `Matcher.find()` and `Matcher.matches()`:

```java
Pattern p = Pattern.compile("\\d+");
Matcher m = p.matcher("abc123def");

System.out.println(m.find());      // true (finds "123")
System.out.println(m.matches());   // false (entire string not digits)
```

Understanding which operation your language provides by default is critical. Python's `re.match()` function, for example, matches from the beginning of the string but does not require matching the entire string: it is neither a full search nor a full match. This subtle behavior causes confusion frequently.

Testing Patterns Interactively

Before integrating regex into production code, you should test patterns interactively to verify they behave as expected. Several approaches exist:

Online regex testers: Tools like regex101.com provide real-time pattern testing with syntax highlighting, match visualization, and engine selection. These are invaluable for learning because they show exactly which parts of your input match and explain each component of the pattern. When using online testers, always select the correct engine flavor (PCRE2, Python, Go, Java, etc.) because patterns can behave differently across engines.

Command-line tools: On Unix-like systems, `grep` is the classic regex testing tool:

```
echo "The quick brown fox" | grep -o "fox"
# Output: fox

echo "Error: connection failed at 02:34:56" | grep -oP '\d{2}:\d{2}:\d{2}'
# Output: 02:34:56
```

The `-o` flag shows only the matched portion, and `-P` enables PCRE syntax. The `ripgrep` tool (`rg`) is a modern alternative that is significantly faster and supports both RE2 and PCRE2 modes.

Language-specific REPLs: Running code interactively in Python's interpreter, Perl's `-e` flag, or a language-specific REPL lets you test regex patterns with actual code:

```
python3 -c "import re; print(re.findall(r'\d+', 'abc123def456'))"
# Output: ['123', '456']

perl -e 'print "$&\n" while "abc123def456" =~ /(\d+)/g'
# Output: 123
#         456
```

Integrated development environment support: Most modern IDEs (VS Code, IntelliJ IDEA, PyCharm) have built-in regex find-and-replace with live preview, which lets you test patterns against your actual codebase or data files before committing to them.

The best approach combines all of these: use an online tester for initial pattern design and learning, command-line tools for quick checks against real data, and language-specific testing for final verification that the pattern works correctly within your target environment. Never assume a pattern works in production without testing it against representative input data, including edge cases like empty strings, very long inputs, unusual character combinations, and text from different languages if internationalization is relevant.

Practical notes:

- Literal matching is case-sensitive by default in all major engines. The pattern `Fox` will not match "fox" unless you enable a case-insensitive flag (discussed later).
- Some languages require escaping backslashes in string literals. In Java and C#, the pattern `\d+` must be written as `"\\d+"` because the backslash is itself an escape character in those languages' string syntax. Python raw strings `r"\d+"` avoid this problem.
- A regex pattern that matches nothing (for example, searching for "xyz" in text that does not contain it) returns a clear no-match result; it does not cause errors or exceptions under normal circumstances.

Chapter 3: Metacharacters and Escaping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

The Dot Metacharacter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Special Characters That Need Escaping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Escape Sequences for Common Characters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

When Escaping Behavior Differs Across Engines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 4: Character Classes and Ranges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Defining Character Sets with Square Brackets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Ranges and Ordering Within Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Negated Character Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Shorthand Character Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

POSIX Character Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 5: Quantifiers and Repetition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

The Asterisk, Plus, and Question Mark

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Exact and Range Quantifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Greedy Matching Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Common Quantifier Mistakes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 6: Anchors and Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Start and End of String Anchors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Word Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Line Anchors and Multiline Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Zero-Width Assertions vs Consuming Characters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 7: Alternation and Grouping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

The Pipe Operator for Alternation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Capturing Groups with Parentheses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Non-Capturing Groups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Prioritizing Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 8: Backreferences and Named Groups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Numeric Backreferences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Named Capture Groups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Backreferences in Replacement Strings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Engine Support Variations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 9: Lookahead and Lookbehind Assertions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Positive Lookahead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Negative Lookahead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Positive Lookbehind

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Negative Lookbehind

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Fixed-Length vs Variable-Length Lookbehind

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 10: Greedy, Lazy, and Possessive Quantifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Understanding Greedy Behavior Step by Step

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Lazy (Reluctant) Quantifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Possessive Quantifiers and Their Performance Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Choosing the Right Quantifier Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 11: Atomic Groups and Advanced Assertions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Atomic Groups and Backtracking Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Conditional Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Recursive and Balanced Matching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Engine-Specific Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 12: Unicode and International Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Unicode Code Points vs Bytes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Unicode Property Escapes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Case Insensitivity Across Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Grapheme Clusters and Combining Characters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 13: Matching, Searching, Extracting, Replacing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Full Match vs Search vs Find All

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Extracting Captured Groups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Splitting Strings on Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Replacement with Backreferences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 14: Real-World Pattern Recipes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Email Address Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

URL and URI Matching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Date and Time Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

File Paths and Filenames

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Log Line Parsing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 15: Source Code and Structured Text Processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Tokenizing Source Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Parsing Configuration Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Working with Markup Languages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

When to Use a Parser Instead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 16: Debugging and Testing Regex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Interactive Testing Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Writing Unit Tests for Regex

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Step-by-Step Debugging Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Common Pitfalls and How to Spot Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 17: Performance and Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Understanding Backtracking Cost

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Catastrophic Backtracking Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Regular Expression Denial of Service (ReDoS)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Writing Linear-Time Patterns with RE2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 18: Maintainability, Portability, and Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Readable Regex: Comments and Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Porting Patterns Across Engines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

When Not to Use Regular Expressions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Documentation and Team Standards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 19: Regex Engines and Flavors Compared

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

NFA vs DFA vs Hybrid Engines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Engine Feature Comparison Table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Unicode Handling Differences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Performance Characteristics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 20: Language-Specific API Guides

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Python (re and regex modules)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Perl (m//, s///, qr//)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Ruby (Regexp class)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Go (regexp package)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Rust (regex crate)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

C (POSIX regex.h and PCRE2)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

C++ (std::regex)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Java (java.util.regex)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

C# (.NET System.Text.RegularExpressions)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Chapter 21: Putting It All Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

The Regex Decision Framework

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Building Your Pattern Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Continuing Your Learning Journey

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Final Thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Glossary of Terms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Syntax and Metacharacter Cheat Sheet

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Core Metacharacters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Quantifiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Shorthand Character Classes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Common Escape Sequences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Common Patterns Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Validation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Extraction Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Transformation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Troubleshooting Guide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Pattern does not match expected input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Pattern matches too much (greedy problem)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Pattern is very slow or hangs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Pattern works in one language but not another

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Unicode-related issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Further Reading and Bibliography

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Books

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Papers and Technical Articles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Official Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Standards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

Online Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringregularexpressions.