Regular Expressions in Practice

The Art of Computer Programming

Pierre Senellart

PSL
UNIVERSITÉ PARIS

January 2026

Outline

Motivation

- Many programming tasks involve finding, validating, or transforming text:
  - extract emails / dates / identifiers from logs
  - validate user input (simple formats)
  - rewrite text (cleanup, normalization)
  - filter lines in command-line pipelines
- Regular expressions (regexps or regexes) are a compact tool to describe patterns in strings
- They are everywhere: editors, IDEs, shells, grep/sed/awk, programming languages, web servers, log tools...

What Is a Regular Expression?

- A regular expression is a pattern language that describes a set of strings
- Typical use modes:
  - match: does this string (or substring) fit the pattern?
  - search: find the first occurrence in a larger text
  - extract: capture parts of what matched
  - replace: rewrite using the captured parts
- Conceptual link (mentioned, not studied here):
  - classical regular expressions correspond to regular languages captured by finite-state automata (FSAs), a very important concept in theoretical computer science
  - but many practical engines also include features that go beyond pure regular languages

Regexps as a DSL

- A regexp is a domain-specific language (DSL) embedded into:
  - command-line tools (`grep`, `sed`)
  - programming languages (`re` in Python, `<regex>` in C++)
  - text editors (VS Code, Vim, etc.)
- Properties of a DSL:
  - specialized syntax optimized for a narrow task (pattern matching)
  - dense notation: very expressive but can become hard to read
- Best practice: treat regexps like code
  - name them / comment them / test them
  - keep them small; split or document when they grow

Outline

Syntax Overview (PCRE-style)

- We present PCRE-style syntax (Perl-Compatible Regular Expressions)
- Tools sometimes use different syntaxes, but what we present tends to be a good common ground
- Core building blocks:
  - literals: `abc`
  - character classes: `[a-z]` `[0-9]` `[^,]`
  - repetition: `*` `+` `?` `{m,n}`
  - alternation: `|`
  - grouping: `(...)` and `(?:...)`
  - anchors: `^` and `$`

Character Classes

- Single-character patterns:
  - `.` : any character (often except newline)
  - `[abc]` : `a` or `b` or `c`
  - `[a-z]` : any lowercase letter
  - `[^0-9]` : any character except digits
- Escapes / predefined classes (common in PCRE):
  - `\d` digit, `\D` non-digit
  - `\w` word char (`[A-Za-z0-9_]` in ASCII mode), `\W` non-word
  - `\s` whitespace, `\S` non-whitespace
- Beware locale/Unicode differences across engines

Repetition and Greediness

- Quantifiers:
  - `r*` : 0 or more
  - `r+` : 1 or more
  - `r?` : 0 or 1
  - `r{m,n}` : between m and n
- Default is greedy: match as much as possible
  - Example: `<.*>` on `<a><b>` matches `<a><b>`
- Non-greedy (lazy) variants in PCRE:
  - `*?`, `+?`, `{m,n}?`
  - Example: `<.*?>` on `<a><b>` matches `<a>`

Grouping and Alternation

- Parentheses group subpatterns:
  - `(ab)+` : repeat `ab` as a unit
  - `cat|dog` : alternation (either `cat` or `dog`)
  - precedence: repetition > concatenation > alternation
- Capturing vs non-capturing groups:
  - `(...)` captures a substring for later extraction/reuse
  - `(?:...)` groups without capturing (often faster / clearer)

Anchors and Boundaries

- Anchors:
  - `^` matches the beginning of the string (or line in multiline mode)
  - `$` matches the end of the string (or line)
- Word boundaries (common):
  - `\b` between word and non-word characters
  - `\B` not a word boundary
- Validation pattern often uses anchors:
  - `^[0-9]+$` the whole string must be digits

How Most Regexp Engines Work: Backtracking

- Many practical regexp engines are backtracking engines
- Intuition:
  - the engine tries to match the pattern left-to-right
  - when it hits a choice, it picks one and remembers alternatives
  - if the match later fails, it backtracks and tries another alternative
- Sources of "choices":
  - alternation: `(cat|car|cap)`
  - optional parts: `u?`
  - repetitions: `.*`, `a+`, `{m,n}`
- Consequences:
  - very flexible feature set (captures, lazy quantifiers, lookarounds...)
  - but performance can degrade badly on some patterns/inputs

Catastrophic Backtracking: A Classic Example

- Some patterns create an exponential number of ways to match
- Example pattern (ambiguous nested repetition):

Text

```
^(a+)+$
```

- "Bad" input: a long run of a's followed by a non-a:

Text

```
aaaaaaaaaaaaaaaaaaaaaaaa!
```

- What happens:
  - `a+` can match the a's in many possible chunkings
  - the final `!` makes the overall match fail
  - the engine tries many decompositions before concluding "no match"
- Practical advice:
  - avoid nested ambiguous quantifiers like `(.*)*`, `(a+)+`
  - make patterns more specific; anchor when possible (`^`, `$`)
  - prefer negated classes over `.*` when you know a delimiter (e.g., `[^,]*` instead of `.*` up to a comma)

Backreferences: Matching What Was Matched Before

- A backreference refers to a previously matched capturing group
- Syntax:
  - `( ... )` captures a subpattern
  - `\1`, `\2`, ... refer to earlier captures (or sometimes `$1`, `$2`, ..., depending on the tool)
- Example: match a repeated word

Text

```
\b(\w+)\s+\1\b
```

- Matches:
  - `the the`
  - `hello hello`
- Does not match:
  - `the a`
  - `hello world`
- Key idea:
  - the second occurrence must be identical to the first
  - this comparison happens at runtime

Backreferences: Power and Consequences

- Backreferences make regexps more powerful than regular languages
  - they cannot be implemented by finite-state automata
  - matching requires remembering arbitrary substrings
- Practical consequences:
  - regexp engines often forced to use backtracking
  - matching can be much slower than expected
- Design guideline:
  - use backreferences when they express intent clearly
  - avoid them in performance-critical or adversarial contexts
  - if possible, validate structure first, then compare explicitly in code

Lookarounds

- Lookarounds assert context without consuming characters
  - `(?=...)` positive lookahead
  - `(?!...)` negative lookahead
  - `(?<=...)` positive lookbehind (engine-dependent restrictions)
  - `(?<!...)` negative lookbehind
- Example: match `foo` only if followed by `bar`
  - `foo(?=bar)`
- Powerful, but can harm readability; use deliberately

Practical Pitfalls

- Escaping rules: regexp syntax + language string syntax
  - the pattern `\d+` may need to be written in some contexts `"\\d+"`
- Match vs search:
  - "does the entire string match?" vs "is there a matching substring?"
- Catastrophic backtracking:
  - some patterns + some engines can be exponentially slow
  - typical danger: nested ambiguous quantifiers
- Do not parse HTML (or other languages with nesting) with regexps (except very constrained cases)
- Easy to design wrong/imprecise patterns (e.g., emails)

Regular Expression Flags (Modifiers)

- Flags modify how a regular expression is interpreted
- They do not change the pattern itself, but its matching rules
- Syntax depends on the environment:
  - command-line tools: options (e.g., `-i`)
  - programming languages: extra arguments
- Common flags:

`i` (IGNORECASE) case-insensitive matching
`m` (MULTILINE) `^` and `$` match line boundaries
`s` (DOTALL) `.` matches newline characters
`x` (EXTENDED) ignore whitespace and allow comments

MULTILINE vs DOTALL: A Common Confusion

- These two flags affect different things
- MULTILINE (`m`):
  - changes the meaning of `^` and `$`
  - they match start/end of each line, not just the whole string
- DOTALL (`s`):
  - changes the meaning of `.`
  - `.` matches newline characters
- Rule of thumb:
  - `m` affects anchors
  - `s` affects wildcards

Outline

Regexps on the Command Line

- Command-line matching is a fast workflow:
  - filter lines from logs
  - locate patterns in a codebase
  - quick validation during development
- `grep` is a commonly used tool to find patterns on the "Unix-like (Linux/macOS/WSL) command line
- Family of grep "dialects":
  - `grep` uses basic regular expressions (BRE) by default
  - `grep -E` uses extended regular expressions (ERE)
  - `grep -P` uses PCRE (if available on your system)

Basic Usage: grep

Shell

```
# Find lines containing "error"
grep "error" server.log

# Case-insensitive (-i) and show line numbers (-n)
grep -in "error" server.log

# Recursively search a directory (-R)
grep -R "TODO" src/
```

- `grep` prints matching lines
- Options are composable; learn a small set and reuse them

Extended vs PCRE: grep -E and grep -P

Shell

```
# BRE: backslashes required for most operators
grep 'cat\|dog' animals.txt

# ERE: alternation without backslashes
grep -E "cat|dog" animals.txt

# PCRE: use \d, \w, lookarounds, etc.
grep -P "^\w+\s+\d+$" data.txt
```

- -E is widely portable
- -P is convenient but may be missing or incomplete on some systems

Useful grep Options

Shell

```
# Show only the matching part
grep -Po "\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b" mail.txt

# Count matches per file / input
grep -c "pattern" file.txt

# Invert match (show non-matching lines)
grep -v "DEBUG" server.log

# Show context lines
grep -n -C 2 "panic" server.log
```

- `-o` is great for "extract" workflows (not just filtering)

Outline

Regexps in Python

- Python provides the `re` module
- Core operations:

`re.search`	find first match anywhere
`re.match`	match at the beginning (often surprising)
`re.fullmatch`	whole string must match
`re.findall`	extract all matches
`re.finditer`	extract all matches, with details for each match
`re.sub`	replace

- Most functions accept `flags=` (`re.IGNORECASE`, `re.MULTILINE`, `re.DOTALL`...)

Regexps in Python: `re` and Raw Strings

- Patterns are written as strings:
  - Important: backslashes are special in *both* Python strings and regexps
- Raw strings: `r"..."`
  - `r"..."` tells Python: do not interpret backslashes as escapes
  - So `r"\d+"` means the two characters `\` and `d`, which the regexp engine understands as "a digit"
  - Without `r""`, you often need to double backslashes: `"\\d+"`

Python

```
import re

re.search(r"\d+", "abc123def")      # OK: regexp sees \d+
re.search("\\d+", "abc123def")      # Equivalent, but harder to read

# Typical gotcha without r"":
re.search("\bword\b", "a word")     # \b is BACKSPACE in Python
↪  strings
re.search(r"\bword\b", "a word")    # \b is WORD BOUNDARY in regexp
```

- Almost always use `r""` for regexps

Search vs Full Match (Python)

Python

```python
import re

s = "ID=1234; user=alice"

m1 = re.search(r"\d+", s)          # finds "1234"
m2 = re.fullmatch(r"\d+", s)       # None (whole string is not digits)

print(m1.group(0))
print(m2)
```

- Prefer `fullmatch` for validation tasks
- Prefer `search` for extraction tasks

Capturing Groups and Named Groups (Python)

Python

```python
import re

text = "alice@example.org"
m = re.fullmatch(r"(?P<user>[^@]+)@(?P<host>.+)", text)

print(m.group(0))            # whole match
print(m.group("user"))       # "alice"
print(m.group("host"))       # "example.org"
print(m.groupdict())         # {'user': ..., 'host': ...}
```

- Named groups make extraction code self-documenting

Finding All Matches (Python)

Python

```python
import re

log = "t=1ms t=12ms t=7ms"
times = [int(m.group(1)) for m in re.finditer(r"t=(\d+)ms", log)]
print(times)   # [1, 12, 7]
```

- `finditer` gives match objects (positions, groups, etc.)
- `findall` is shorter but behaves differently with groups

Replacing with re.sub (Python)

Python

```python
import re

s = "user: alice, id: 1234"
t = re.sub(r"\bid:\s*(\d+)\b", r"id:<\1>", s)
print(t)  # user: alice, id:<1234>
```

- Replacement strings can refer to groups (e.g., \\1)
- For complex logic, sub can take a callback function

Flags and Compilation (Python)

Python

```python
import re

pat = re.compile(r"^\s*#.*$", flags=re.MULTILINE)
text = "x\n# comment\n y\n"

print(pat.findall(text))  # ['# comment']
```

- Compile regexps you reuse: clearer + may be faster

Outline

Regexps in C++11

- C++11 standard library provides `<regex>`
- Main types:
  - `std::regex` compiled pattern
  - `std::smatch` match results for `std::string`
- Main operations:
  - `std::regex_search`: find a match anywhere
  - `std::regex_match`: whole string must match
  - `std::regex_replace`: substitution
- The standard supports multiple grammars; we will target ECMAScript by default (closest to PCRE style, but not all features available)

Search and Extract (C++11)

C++

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

int main() {
  std::string s = "ID=1234; user=alice";
  std::regex r(R"(ID=(\d+))");   // raw string: fewer escapes

  std::smatch m;
  if (std::regex_search(s, m, r)) {
    std::cout << "whole: " << m[0] << "\n"; // "ID=1234"
    std::cout << "group: " << m[1] << "\n"; // "1234"
  }
}
```

- Prefer raw strings `R"( ... )"` for readability (note the use of both double quotes and parentheses)

Whole-String Validation (C++11)

C++

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

int main() {
  std::regex digits(R"(\d+)");
  std::cout << std::regex_match("12345", digits) << "\n"; // 1
  std::cout << std::regex_match("12a45", digits) << "\n"; // 0
}
```

- `regex_match` checks the whole string (like Python `fullmatch`)

Replacement (C++11)

C++

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

int main() {
  std::string s = "id: 1234, id: 7";
  std::regex r(R"(\bid:\s*(\d+)\b)");
  std::string t = std::regex_replace(s, r, "id:<$1>");
  std::cout << t << "\n"; // id:<1234>, id:<7>
}
```

- Replacement uses $1, $2, ... for groups

Practical Notes for C++

- Engine differences:
  - default grammar is ECMAScript; not full PCRE
  - some advanced PCRE features (lookbehind, etc.) are not portable in `<regex>`
- Performance considerations:
  - compile `std::regex` once if you reuse it
  - be mindful of backtracking-heavy patterns
- For PCRE-level features in C++:
  - external libraries exist (PCRE2, RE2), but are out of scope here

Outline

Choosing the Right Tool

- Regexps are great when:
  - the structure is local and regular (identifiers, simple tokens, log formats)
  - you need quick extraction or filtering
- Regexps are a bad fit when:
  - you need nested structure (general parsing)
  - readability and maintainability are an issue
- Best practices:
  - keep patterns small; build with helper code when needed
  - write tests for non-trivial patterns

Summary

- Regexps describe sets of strings; widely used for matching, extraction, replacement
- They are a DSL embedded into many tools and languages
- We used PCRE-style syntax as a reference, but engines differ in details
- Command line: `grep`, `grep -E`, `grep -P`
- Python: `re.search`, `re.fullmatch`, `re.finditer`, `re.sub`
- C++11: `std::regex_search`, `std::regex_match`, `std::regex_replace`

Licensing

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.