

chapter one

What a Regular Expression Is

A regex can fit a large amount of syntax into a short line:

```
^(?:[a-z0-9!#$%&'*/+=?^_`{|}~]+)@
(?:[a-z0-9-]+\.)+[a-z]{2,}$
```

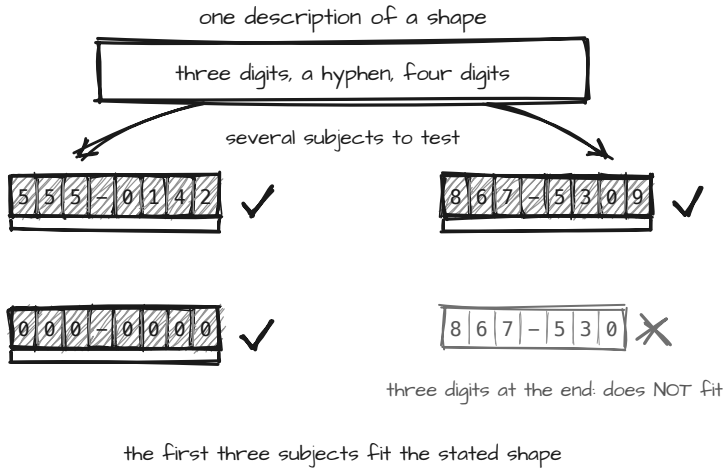
A long pattern is still
a sequence of tokens.

This example is easier to read after it is split into tokens. The `^` and `$` constrain positions, bracketed expressions describe allowed characters, `+` means repetition, and `@` is literal text. This chapter first defines the values involved in matching; later chapters explain each token.

A pattern describes strings

One pattern can
accept many
different strings.

A regular expression is a pattern interpreted by a regex engine. In a full-match operation, the pattern describes a set of strings. A phone-number pattern might describe three digits, a hyphen, and four digits. That set includes `555-0142`, `867-5309`, and `000-0000`; it excludes `867-530`, which has only three digits after the hyphen.



Three subjects have the required three-digit, hyphen, four-digit shape. The shorter fourth subject does not.

The pattern is the description, not the example used to explain it. A search operation applies the same description to substrings within a larger subject. For example, a pattern describing `cat` can locate those three consecutive letters inside `The cat sat on the mat.` without matching the whole sentence.

“Regular expression” covers several related pattern languages. The classical definition describes regular languages, while widely used engines add features such as backreferences that go beyond that definition. This book uses *regex* for the practical family and names important dialect differences as they arise.

Pattern, subject, engine, and result

A match operation involves a **pattern**, a **subject**, and an **engine**. The pattern states the condition. The subject is the text being examined. The engine is the implementation that interprets the pattern.

The scan is a result-oriented model, not an implementation promise.

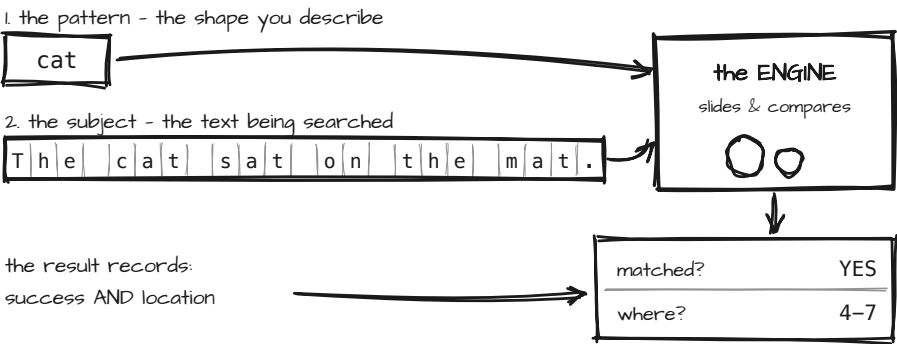
A common search model starts at the left edge of the subject and attempts the pattern at successive positions until an attempt succeeds or no position remains. This model accurately explains the examples in the early chapters, but it is not a trace of every implementation. Engines can compile, combine, or optimize attempts while returning the same result.

The result has at least two relevant parts: whether a match exists and, if it does, the span it occupies. APIs may also return captures and other metadata.

Value	Meaning	Example
Pattern	Condition to match	cat
Subject	Text being examined	The cat sat on the mat.
Result	Success and location	match at offsets 4–7

This book writes spans as start-inclusive and end-exclusive.

Offsets here use the common half-open convention: the start is included and the end is excluded. The span 4–7 therefore contains the characters at zero-based offsets 4, 5, and 6. An API may report the same location in a different form.



The pattern and subject enter a matching operation. Its result records success and the matched span.

A literal pattern

Consider the subject:

The cat sat on the mat.

and the pattern:

cat

These three pattern characters are literals: `c`, followed by `a`, followed by `t`. A search attempt fails at the opening `T` and at the next positions. It succeeds when the start position reaches the `c` at offset 4.

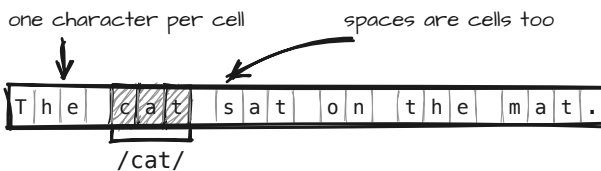
The book marks a consumed span with carets beneath the subject:

The cat sat on the mat.

^^^

/cat/ matches here

Each cell represents one character, including a space.



the matched span contains these three cells

The subject occupies one cell per character. Hatching and the lower bracket mark the three-character span consumed by `cat`.

The carets cover one contiguous, ordered span. Other `a` and `t` characters in the sentence cannot be combined with the earlier `c`; separate characters do not satisfy the literal sequence `cat`.

A match span is contiguous; scattered characters do not combine.

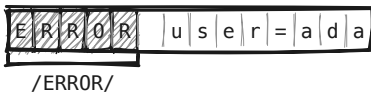
Common operations

Regex APIs expose several related operations:

Search, validation,
extraction, and
replacement return
different results.

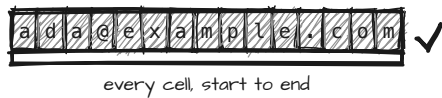
- **Search** locates a matching span within a subject.
- **Full match or validation** asks whether the pattern covers the complete subject, often with additional application checks.
- **Extraction** returns the full match or selected captured spans.
- **Replacement** constructs new text from fixed material, matches, and captures.

1. SEARCH - locate a span



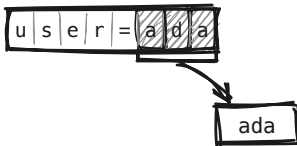
output: a location

2. VALIDATE - check coverage



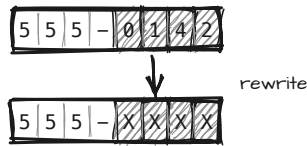
output: yes or no

3. EXTRACT - return the span



output: text

4. REPLACE - rewrite the span



output: a new string

Four common operations return different forms of output: a location, a decision, extracted text, or a rewritten string.

Editors, command-line tools, and programming-language libraries provide different combinations of these operations. Their pattern syntax overlaps, but it is not interchangeable without checking the dialect, flags, API, and host-language string escaping.

Search and full match are different questions

Searching `The cat sat on the mat.` for `cat` succeeds because the pattern matches the substring at offsets 4–7. Requiring the same pattern to match the complete subject fails because the subject contains 23 characters, not three.

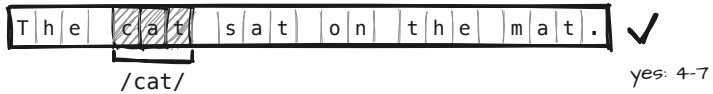
The cat sat on the mat.

^^^

search: match at offsets 4–7

Search locates a span; full match constrains the complete subject.

SEARCH: where inside this string does it fit?



FULL MATCH: is the WHOLE string exactly this?



no: 23 chars, not 3

the operation changes the coverage requirement

A search finds `cat` within the sentence. A full-match operation rejects the sentence because the pattern covers only three of its 23 characters.

The operation is part of the specification. Python, for example, exposes `search`, `match`, and `fullmatch` with different start and coverage rules. Other environments use anchors, API options, or separate methods. When a pattern appears correct but a call returns an unexpected result, inspect the operation before changing the pattern.

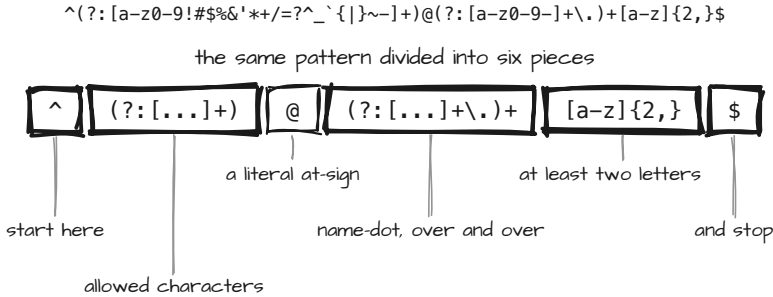
Check the API operation as well as the pattern.

Reading a pattern as tokens

The opening example combines anchors, groups, classes, quantifiers, and literals. Dividing it into those tokens exp-

Token boundaries
reveal the structure
of a dense pattern.

uses the job of each part:



token boundaries expose the pattern structure

The opening pattern divided into six functional pieces: start anchor, the allowed characters, a literal at-sign, name-dot over and over, at least two letters, and end anchor.

That decomposition does not make the example a complete email-address validator. It only makes its stated behavior readable. Correctness still depends on the intended inputs, the selected dialect, and tests at the pattern's boundaries.

The next chapter traces literal search attempts one position at a time.

Exercises

Exercise 1.1. In plain English, describe the strings accepted by the phone-number shape “three digits, a hyphen, four digits.” Give two matching strings other than `555-0142` and one nonmatching string.

Exercise 1.2. Identify the pattern, subject, and result when searching `The cat sat on the mat.` for `cat`. Include both parts of the result.

Exercise 1.3. Apply `cat` to `The cat sat on the mat.` first as a search and then as a full-match operation. Explain why the results differ.

Exercise 1.4. The letters `c`, `a`, and `t` occur elsewhere in the sentence. Explain why those occurrences do not form another match for the literal pattern `cat`, and report the successful match using half-open offsets.