

COURSE

Regular Expressions in Practice

10 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1 Matching Basics 2

Chapter 2 Groups and Captures 7

Chapter 3 Alternation and Flags 13

Chapter 4 Substituting and Splitting 19

Chapter 5 Lookaround 25

Chapter 6 Compiling and Iterating 31

Chapter 7 Validating Real Data 39

Chapter 8 Parsing Logs 46

Chapter 9 Pitfalls 52

Chapter 10 Text Toolkit 59

Regular Expressions in Practice

Chapter 1 · Matching Basics

A **regular expression** (regex) is a small pattern language for describing text: "four digits, a dash, two digits", "a word that starts with a capital", "anything between angle brackets". Instead of writing a loop that inspects characters one at a time, you describe the shape of what you want and let the engine find it. This course is practical: each chapter takes one idea, runs it, and ends with a rule you can reuse. All examples use Python 3 and its built-in `re` module.

> **WHAT WAS RUN**
>
> Every snippet in this chapter was executed, and the output shown in the comments is what Python printed. Run them yourself in a file or the interactive prompt; there is nothing to install.

Three Ways to Ask "Does It Match?"

The `re` module has three entry points that differ only in *where* the pattern is allowed to match. Picking the wrong one is the most common beginner mistake, so learn the difference first.

FUNCTION	WHERE IT LOOKS	RETURNS
re.search	anywhere in the string	first match, or `None`
re.match	only at the start of the string	match, or `None`
re.fullmatch	the whole string, start to end	match, or `None`
re.findall	everywhere	a list of every non-overlapping match

```
import re

text = "Order 4821 shipped on 2026-11-02 to bay 7."

print(re.search(r"\d+", text))      # <re.Match object; span=(6, 10), match='4821'>
print(re.match(r"\d+", text))       # None  (the text starts with "Order", not a digit)
print(re.fullmatch(r"\d+", "4821"))  # <re.Match object; span=(0, 4), match='4821'>
```

```
m = re.search(r"\d+", text)
print(m.group(), m.start(), m.end(), m.span())  # 4821 6 10 (6, 10)
```

A successful call returns a **match object**; a failed one returns `None`. Because `None` is falsy, `if re.search(...):` is the usual test. On the match object, `.group()` is the matched text and `.span()` gives the start and end positions.

Raw Strings: Always Write r"..."

escapes like `\n`. If you write the pattern as an ordinary string, Python processes the backslashes first and can change your meaning. A **raw string**, written with an `r` prefix, leaves every backslash alone and hands it to the regex engine untouched.

```
print(len("\n"), len(r"\n"))     # 1 2   (one newline character versus a backslash and a

print(re.search("\\d+", "a12").group())   # 12   (works, but the doubled backslash is ea
print(re.search(r"\d+", "a12").group())   # 12   (the same pattern, readable)
```

The Building Blocks

Most patterns are assembled from a few kinds of piece. Most characters simply match themselves; a handful are special.

PIECE	MEANING	EXAMPLE
`\d` `\w` `\s`	a digit; a letter, digit or underscore; whitespace	`\d\d` matches `42`
`.`	any one character except a newline	`c.t` matches `cat`, `cot`
`[abc]`	one character from the set	`[aeiou]` matches a vowel
`[^abc]`	one character *not* in the set	`[^0-9]` matches a non-digit

PIECE	MEANING	EXAMPLE
`*` `+` `?`	zero or more; one or more; zero or one	`colou?r` matches both spellings
`{n}` `{n,m}`	exactly n; between n and m repeats	`\d{4}` matches four digits
`^` `$`	start of the string; end of the string	`^cat$` matches only `cat`
`\b`	a word boundary (between a word character and a non-word one)	`\bcat\b` matches the word, not `concat`

```
print(re.findall(r"\d+", text))                 # ['4821', '2026', '11', '02', '7']
print(re.findall(r"\d{4}-\d{2}-\d{2}", text))   # ['2026-11-02']
print(re.findall(r"[aeiou]", "regular"))         # ['e', 'u', 'a']
print(re.findall(r"[^aeiou\s]+", "regular expressions"))
# ['r', 'g', 'l', 'r', 'xpr', 'ss', 'ns']
```

Note the last line: `[^aeiou\s]+` means "one or more characters that are neither a vowel nor whitespace", so the engine returns each run of consonants it finds.

Anchors and Word Boundaries

Anchors match a *position*, not a character. Without them, a pattern is happy to match inside a longer word, which is rarely what you want.

```
for s in ["cat", "concat", "cats"]:
    print(s, bool(re.search(r"^cat$", s)),
             bool(re.search(r"\bcat\b", s)),
             bool(re.search(r"cat", s)))
# cat True True True
# concat False False True
# cats False False True
```

Greedy Versus Lazy

Quantifiers such as `*` and `+` are **greedy**: they grab as much text as they can while still letting the overall pattern succeed. Adding a `?` after the quantifier makes it **lazy**, taking as little as possible. The classic trap is matching tags.

```
html = "<b>bold</b> and <i>italic</i>"

print(re.findall(r"<.*>", html))    # ['<b>bold</b> and <i>italic</i>']   one huge match
print(re.findall(r"<.*?>", html))   # ['<b>', '</b>', '<i>', '</i>']
```

The greedy `.*` ran to the very last `>` in the string. The lazy `.*?` stopped at the first one. Often a better fix than laziness is a more precise class, such as `<[^>]*>`, which says "anything except a closing bracket".

> **A HABIT WORTH FORMING**
>
> Choose the function by the question you are asking: `fullmatch` to validate a whole value, `search` to find something inside text, `findall` to collect every occurrence. Always write the pattern as a raw string.

Challenges

EXERCISE 1

Given `log = "disk 87% full, cpu 5%, mem 100%"`, write one `findall` call that returns only the numbers (`['87', '5', '100']`) and a second that returns them with the percent sign (`['87%', '5%', '100%']`). Hint: the first can use a lookahead, `(?=%)`, which checks that a percent sign follows without including it.

View solution

EXERCISE 2

Write `is_code(s)` that returns `True` only for a product code made of exactly three capital letters, a dash and four digits (such as `ABC-1234`). Test it on `"AB-1234"`, `"ABC-12345"`, `"abc-1234"` and `"xABC-1234"`. Then show what `re.match` does with `"ABC-12345"` and explain why `fullmatch` is the right choice.

View solution

EXERCISE 3

For `tags = "<p>one</p><p>two</p>"`, show that `<p>.*</p>` returns one big match. Fix it twice: once with a lazy quantifier and once with a negated class. Predict the output before you run each.

View solution

QUICK REFERENCE

- **search** anywhere; **match** at the start; **fullmatch** the whole string; **findall** every match
- **Failure:** returns `None`, so test with `if`
- **Always** write patterns as raw strings: `r"\d+"`
- **Classes:** `\d \w \s`, sets `[abc]`, negated sets `[^abc]`
- **Repeats:** `* + ? {n,m}` are greedy; add `?` for lazy
- **Positions:** `^ $ \b` match places, not characters

COMING NEXT

Regular Expressions in Practice 2 introduces groups: capturing parts of a match, naming them, and using them to pull a date or an email address apart into its pieces.

Regular Expressions in Practice

Chapter 2 · Groups and Captures

So far a match has been one lump of text. Real work usually needs the *pieces*: the year, month and day of a date, the user and host of an address. Parentheses turn part of a pattern into a **group**, and the engine remembers what each group matched. This chapter covers numbered groups, named groups, groups that do not capture, and how `findall` changes its answer when groups appear.

> **WHAT WAS RUN**
>
> As in Regular Expressions in Practice 1, every snippet was executed and the comments show what Python printed.

Numbered Groups

Each pair of parentheses is a group, numbered from 1 by the position of its opening bracket, reading left to right. Group 0 is always the whole match.

```
import re

text = "Booked 2026-11-02 and 2027-01-15."
m = re.search(r"(\d{4})-(\d{2})-(\d{2})", text)

print(m.group(), m.group(0))                # 2026-11-02 2026-11-02
print(m.group(1), m.group(2), m.group(3))   # 2026 11 02
print(m.groups())                           # ('2026', '11', '02')
print(m.span(2))                            # (12, 14)

year, month, day = m.groups()
print(year, month, day)                     # 2026 11 02
```

`.groups()` returns every group except group 0 as a tuple, which makes unpacking into variables a one-liner. `.span(n)` gives the position of one group inside the original string. Captured values are always *strings*: convert with `int()` when you need a number.

Nested groups

When groups sit inside one another, count opening brackets from the left. The outer group is numbered before the ones inside it.

```
m = re.search(r"((\d{3})-(\d{4}))", "call 555-0199 now")
print(m.groups())     # ('555-0199', '555', '0199')
```

Groups That Did Not Take Part

A group followed by `?` may match nothing. It still exists, but its value is `None`, not an empty string. Check for that before using it.

```
pat = r"(\w+)@(\w+)\.com(\.uk)?"

m = re.search(pat, "bob@site.com")
print(m.groups(), m.group(3))    # ('bob', 'site', None) None

m = re.search(pat, "bob@site.com.uk")
print(m.groups())                # ('bob', 'site', '.uk')
```

Named Groups

Numbers are fragile: add a group at the front and every number after it shifts. A **named group**, written `(?P<name>...)`, lets you ask for a part by what it means. It still has a number too.

```
m = re.search(r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})", text)

print(m.group("year"), m["month"], m.group("day"))   # 2026 11 02
print(m.groupdict())    # {'year': '2026', 'month': '11', 'day': '02'}
print(m.group(1))          # 2026   (the number still works)
```

`.groupdict()` returns a dictionary of name to text, which is ideal for building records. The shorthand `m["month"]` works the same as `m.group("month")`. Together with `finditer` you can turn a line of text into data:

```
line = "alice=30, bob=25"
pat = r"(?P<name>\w+)=(?P<age>\d+)"

for m in re.finditer(pat, line):
    print(m.groupdict())
# {'name': 'alice', 'age': '30'}
# {'name': 'bob', 'age': '25'}

print({m["name"]: int(m["age"]) for m in re.finditer(pat, line)})
# {'alice': 30, 'bob': 25}
```

Non-Capturing Groups

Sometimes you need parentheses only to *group*: to apply a quantifier to several characters, or to limit an alternation. If you do not want the text remembered, write `(?:...)`. It takes no number and does not appear in `.groups()`.

SYNTAX	GROUPS?	CAPTURES?	USE IT FOR
`( ... )`	yes	yes, numbered	pieces you want back
`(?P<name> ... )`	yes	yes, named and numbered	pieces with a meaning
`(?: ... )`	yes	no	repeating or choosing, nothing to keep

```
people = "Dr. Lee met Ms. Wu and Mr. Fox"

print(re.findall(r"(?:Mr|Ms|Dr)\. (\w+)", people))   # ['Lee', 'Wu', 'Fox']
print(re.findall(r"(Mr|Ms|Dr)\. (\w+)", people))     # [('Dr', 'Lee'), ('Ms', 'Wu'), ('M
```

How findall Reacts to Groups

This is the rule that surprises people. What `findall` returns depends on how many capturing groups the pattern has:

CAPTURING GROUPS	FINDALL RETURNS
none	a list of the whole matches
exactly one	a list of that group's text
two or more	a list of tuples, one item per group

```
print(re.findall(r"\d{4}-\d{2}-\d{2}", text))          # ['2026-11-02', '2027-01-15']
print(re.findall(r"(\d{4})-\d{2}", text))              # ['2026', '2027']
print(re.findall(r"(\d{4})-(\d{2})-(\d{2})", text))
# [('2026', '11', '02'), ('2027', '01', '15')]

hosts = "hosts 10.0.0.1 and 192.168.1.20"
print(re.findall(r"(?:\d{1,3}\.){3}\d{1,3}", hosts))   # ['10.0.0.1', '192.168.1.20']
print(re.findall(r"(\d{1,3}\.){3}\d{1,3}", hosts))     # ['0.', '1.']
```

The last two lines are the classic accident. Adding a capturing group just to repeat it changes the result to the *last repetition of that group* (`'0.'` and `'1.'`), not the address. Use `(?:...)` when the parentheses are only there for repeating. A repeated capturing group always keeps only its final repetition:

```
print(re.fullmatch(r"(\d)+", "123").group(1))   # 3
```

Reusing a Group Inside the Pattern

A **backreference** such as `\1` means "the same text group 1 matched". It lets a pattern find repetition, like a word typed twice in a row.

```
print(re.findall(r"\b(\w+) \1\b", "this is is a test test of the the code"))
# ['is', 'test', 'the']
```

A HABIT WORTH FORMING

Name any group you will read later, make any group you will not read non-capturing, and remember that every captured value is a string until you convert it.

Challenges

EXERCISE 1

Given `s = "Smith, John; Doe, Jane"`, use `findall` with two capturing groups to get the surname and first name of each person as tuples, then build the list `['John Smith', 'Jane Doe']` from them.

View solution

EXERCISE 2

Write `parse_time(t)` using named groups `h`, `m` and an optional `s` for strings like `9:05` and `14:30:59`. Return a dictionary of integers (seconds default to 0) or `None` if the string does not fit. Test it on `"9:05"`, `"14:30:59"`, `"9:5"` and `"1:2:3"`. Hint: the optional seconds need a non-capturing group around the colon.

View solution

EXERCISE 3

For `urls = "http://a.com/x https://b.org/y/z ftp://c.net"`, predict and then run four things: `findall` with a capturing protocol group and a capturing host group; the same with the protocol group made non-capturing; the same with both groups named; and a `finditer` loop printing each `groupdict()`. Explain why the shape of the `findall` result changes.

View solution

QUICK REFERENCE

- **Group:** `( ... )` numbered from 1 by opening bracket; group 0 is the whole match
- **Read back:** `m.group(n)`, `m.groups()`, `m.span(n)`
- **Named:** `(?P<name> ... )`, then `m["name"]` or `m.groupdict()`
- **Non-capturing:** `(?: ... )` groups without remembering
- **Unused group:** value is `None`
- **findall:** strings for 0 or 1 groups, tuples for 2 or more
- **Backreference:** `\1` repeats what group 1 matched

COMING NEXT

Regular Expressions in Practice 3 covers alternation and flags: choosing between patterns with the pipe, ignoring case, matching across lines, and writing long patterns readably.

Regular Expressions in Practice

Chapter 3 · Alternation and Flags

Two tools finish the basics. **Alternation** lets one pattern say "this or that". **Flags** change how the whole engine reads a pattern: ignoring case, letting `^` and `$` work on every line, letting the dot cross line breaks, and allowing you to lay a long pattern out with spaces and comments.

> **WHAT WAS RUN**
>
> As in the earlier chapters, every snippet was executed and the comments show what Python printed.

Alternation with the Pipe

The pipe `|` means "match the left side, or else the right side". It has the *lowest* precedence of any operator, so it splits the whole pattern unless you fence it in with a group.

```
import re

s = "A cat, a dog and a catalog"
print(re.findall(r"cat|dog|bird", s))           # ['cat', 'dog', 'cat']
print(re.findall(r"\b(?:cat|dog|bird)\b", s))   # ['cat', 'dog']
```

The first pattern also found the `cat` inside `catalog`. Wrapping the choice in `(?:...)` and adding `\b` word boundaries on both sides restricts it to whole words.

The pipe splits the whole pattern

Without parentheses, each side of the pipe is a complete alternative, including any anchors or text beside it.

```
print(re.search(r"^a|bc", "xbc").group())      # bc   (means: starts with a, OR contains
print(re.search(r"^(?:a|b)c", "bc") is not None)  # True (starts with a or b, then c)
print(re.findall(r"gr(?:a|e)y", "gray grey graey"))   # ['gray', 'grey']
print(re.findall(r"gr[ae]y|colou?r", "gray color colour"))
```

```
# ['gray', 'color', 'colour']
```

When the choice is a single character, a class such as `[ae]` is shorter and faster than `(?:a|e)`. Use the pipe when the alternatives are longer than one character.

Order matters

The engine tries alternatives *left to right* and takes the first one that lets the whole pattern succeed. It does not look for the longest. Put longer alternatives first when one is a prefix of another.

```
print(re.search(r"cat|catalog", "catalog").group())   # cat
print(re.search(r"catalog|cat", "catalog").group())   # catalog
```

Flags

A flag is passed as a third argument (`re.findall(pattern, text, re.I)`) or, later, to `re.compile`. Combine several with the bitwise-or operator `|`. Each has a long name and a one-letter alias.

LONG NAME	SHORT	EFFECT
`re.IGNORECASE`	`re.I`	letters match in either case
`re.MULTILINE`	`re.M`	`^` and `$` match at every line, not just the string ends
`re.DOTALL`	`re.S`	`.` also matches the newline character
`re.VERBOSE`	`re.X`	whitespace and `#` comments in the pattern are ignored

IGNORECASE

```
print(re.findall(r"hello", "Hello HELLO hello"))                # ['hello']
print(re.findall(r"hello", "Hello HELLO hello", re.IGNORECASE))  # ['Hello', 'HELLO', 'h
```

```
print(re.findall(r"(?i)hello", "Hello HELLO hello"))          # ['Hello', 'HELLO', 'he
```

The last line shows an **inline flag**: `(?i)` at the start of the pattern switches the flag on without an extra argument. The letters are `i`, `m`, `s` and `x`. Inline flags are handy when the pattern is stored somewhere that cannot carry a separate flags value.

MULTILINE

By default `^` matches only at the very start of the string and `$` only at the very end (or just before a final newline). With `re.M` they match at the start and end of *each line*.

```
t = "one\ntwo\nthree"
print(re.findall(r"^\w+$", t))                # []
print(re.findall(r"^\w+$", t, re.MULTILINE))  # ['one', 'two', 'three']
print(re.findall(r"^t\w+", t, re.M))          # ['two', 'three']
```

The first call finds nothing because `^\w+$` would need the whole string to be one word. Note that `\w` never matches a newline, so a match can never leak from one line into the next here.

DOTALL

The dot normally stops at a newline. `re.S` lets it cross, which matters when a single item spans several lines.

```
print(re.search(r"one.two", t))                    # None
print(repr(re.search(r"one.two", t, re.S).group()))    # 'one\ntwo'
print(re.findall(r".+", t))                        # ['one', 'two', 'three']
print(re.findall(r".+", t, re.S))                  # ['one\ntwo\nthree']

src = "<b>x\ny</b> <b>z</b>"
print(re.findall(r"<b>(.*?)</b>", src))            # ['z']
print(re.findall(r"<b>(.*?)</b>", src, re.S))      # ['x\ny', 'z']
```

Be careful not to mix up the two line flags. **MULTILINE changes the anchors; DOTALL changes the dot.** They are independent and are often used together.

Combining flags

```
log = "ERROR disk\nwarn cpu\nInfo ok\nerror net"

print(re.findall(r"^(?:error|warn)\b.*$", log))
# []
print(re.findall(r"^(?:error|warn)\b.*$", log, re.I | re.M))
# ['ERROR disk', 'warn cpu', 'error net']
```

VERBOSE

A long pattern on one line is hard to read. `re.X` tells the engine to ignore whitespace and everything from a `#` to the end of the line, so you can lay a pattern out in parts and explain each part. Use a triple-quoted raw string.

```
pat = re.compile(r"""
    (?P<area>\d{3})   # three-digit area code
    [-.\s]?           # optional separator
    (?P<num>\d{4})    # four-digit number
""", re.VERBOSE)

print(pat.search("call 555-0199").groupdict())   # {'area': '555', 'num': '0199'}
```

The price of verbose mode is that spaces and `#` in the pattern are now ignored, so to match them literally you must escape them (`\ `, `\#`) or put them in a class (`[ ]`, `[#]`).

```
print(re.search(r"a b", "a b").group())            # a b
print(re.search(r"a b", "a b", re.X))              # None  (the space was ignored, so it
print(re.search(r"a[ ]b", "a b", re.X).group())    # a b
print(re.search(r"a#b", "a#b", re.X).group())      # a   ("#b" was treated as a comment)
```

CHOOSING FLAGS

Reach for `re.I` when the data's case is unreliable, `re.M` when you process text line by line as one string, `re.S` when one item crosses lines, and `re.X` for any pattern longer than a single short line.

Challenges

EXERCISE 1

Given `s = "The Cat sat; a DOG barked; cats and dogs; concat"`, find every whole word that is `cat` or `dog`, with an optional plural `s`, ignoring case. Print the list and its length. Then run the naive `cat|dog` with `re.I` and explain the difference.

View solution

EXERCISE 2

Given a four-line log string (each line: a date, a level in mixed case, a message), use `finditer` with named groups `date`, `level` and `msg`, and the flags needed, to print a `groupdict()` for only the `error` and `warn` lines. Then show what `findall` returns for the same pattern when you leave out the flags, and say why.

View solution

EXERCISE 3

Write a verbose pattern with comments that matches clock times such as `9:05`, `14:30:59` and `9:05 PM`: named groups `h`, `m`, optional `s` (non-capturing colon group) and optional `ap`, with leading and trailing space allowed and case ignored. Test it on `"9:05"`, `"14:30:59"`, `"9:05 PM"`, `"9:5"` and `"25:61x"`, and say why the last one does not match.

View solution

QUICK REFERENCE

- **Alternation:** `a|b` has the lowest precedence; fence it with `(?: ... )`

- **Order:** alternatives are tried left to right; put longer ones first
- **Whole words:** `\b(?:cat|dog)\b`
- **re.I:** ignore case; inline `(?i)`
- **re.M:** `^` and `$` work per line
- **re.S:** `.` also matches a newline
- **re.X:** ignore pattern whitespace and `#` comments; escape or bracket literal spaces
- **Combine:** `re.I | re.M`

COMING NEXT

Regular Expressions in Practice 4 covers substituting and splitting: `re.sub` with backreferences and functions, and `re.split`.

Regular Expressions in Practice

Chapter 4 · Substituting and Splitting

So far patterns have only *found* text. This chapter uses them to *change* it. `re.sub` replaces every match with something else, and `re.split` cuts a string wherever a pattern matches. Both are far more flexible than the plain string methods `str.replace` and `str.split`.

> **WHAT WAS RUN**
>
> As in the earlier chapters, every snippet was executed and the comments show what Python printed.

re.sub

The call is `re.sub(pattern, replacement, string, count=0, flags=0)`. It returns a *new* string; the original is never changed. A `count` of 0 means "replace all".

```
import re

print(re.sub(r"\s+", " ", "a   b \t\n c"))          # a b c
print(re.sub(r"\d", "#", "tel 555-0199"))            # tel ###-####
print(re.sub(r"\d", "#", "a1b2c3d4", count=2))       # a#b#c3d4
print(re.subn(r"\d", "#", "a1b2c3"))                 # ('a#b#c#', 3)
```

`re.subn` works the same way but returns a tuple of the new string and the number of replacements, which is useful when you need to know whether anything changed.

Backreferences in the replacement

The replacement string can reuse what the pattern captured. Write `\1`, `\2` for numbered groups, or `\g<name>` for named ones. This is how you reorder text.

```
print(re.sub(r"(\w+), (\w+)", r"\2 \1", "Smith, Ann"))
# Ann Smith
```

```
pat = r"(?P<y>\d{4})-(?P<m>\d\d)-(?P<d>\d\d)"
print(re.sub(pat, r"\g<d>/\g<m>/\g<y>", "on 2026-11-02 and 2025-01-09"))
# on 02/11/2026 and 09/01/2025

print(re.sub(r"(\d+)", r"\g<1>0", "7 and 12"))   # 70 and 120
```

The last line shows why `\g<1>` exists: `\10` would be read as group ten, not group one followed by a zero. Use the `\g<...>` form whenever a digit follows the reference.

Backreferences inside the pattern

A group can also be referred to *within the pattern itself*, to demand a repeat of whatever it matched.

```
print(re.sub(r"(\w)\1", r"<\1\1>", "book keeper tree"))
# b<oo>k k<ee>per tr<ee>
```

Here `(\w)\1` matches a doubled letter. In the pattern, `\1` means "the same character group 1 just matched"; in the replacement, it means "insert group 1".

A function as the replacement

When plain reordering is not enough, pass a *function*. `re.sub` calls it once per match with the match object and substitutes whatever string it returns. The replacement function can do arithmetic, look things up, or format.

```
print(re.sub(r"(\d+)", lambda m: str(int(m.group()) * 2), "3 apples and 10 pears"))
# 6 apples and 20 pears

print(re.sub(r"\b[a-z]", lambda m: m.group().upper(), "the quick brown fox"))
# The Quick Brown Fox

def money(m):
    return f"{int(m['n']):,}"

print(re.sub(r"(?P<n>\d{4,})", money, "pop 1234567 and 9876"))
```

```
# pop 1,234,567 and 9,876
```

The function must return a *string*, which is why the first example wraps the number in `str()`. Inside the function, `m.group()`, `m[1]` and `m['n']` all work as in earlier chapters.

Escapes in the replacement string

The replacement string is itself interpreted: a backslash starts an escape. A literal backslash in the output must be doubled. When literal text containing special characters is used as the *pattern*, protect it with `re.escape`.

```
print(re.sub(r"\\", "/", r"C:\dir\file"))        # C:/dir/file
print(re.sub("a", r"\\", "cat"))                 # c\t
print(repr(re.sub("a", r"\n", "cat")))           # 'c\nt'  (a real newline)

print(re.escape("1+1=2?"))                           # 1\+1=2\?
print(re.sub(re.escape("1+1"), "two", "1+1=2"))   # two=2
```

A function replacement avoids the escape rules altogether, because its return value is used as-is. If a substitution inserts user-supplied text, return it from a function.

Empty matches

A pattern that can match nothing replaces at every position between characters, because an empty match exists everywhere.

```
print(re.sub(r"x*", "-", "abc"))   # -a-b-c-
```

re.split

`re.split(pattern, string, maxsplit=0, flags=0)` returns a list of the pieces between matches. Unlike `str.split`, the separator can be a pattern.

```
print(re.split(r",", "a,b,,c"))                  # ['a', 'b', '', 'c']
print(re.split(r"\s*,\s*", "a , b,c ,  d"))     # ['a', 'b', 'c', 'd']
print(re.split(r"[;,]\s*", "a; b,c", maxsplit=1))  # ['a', 'b,c']
print(re.split(r"\W+", "Hello, world! It's me."))
# ['Hello', 'world', 'It', 's', 'me', '']
```

Two things to notice. Consecutive separators produce an empty string between them (`'a,b,,c'`). And a separator at the very end leaves an empty string after it, as in the last line; filter these out with `[p for p in parts if p]` when you do not want them.

Keeping the separators

If the split pattern contains a capturing group, the text it matched is included in the result between the pieces.

```
print(re.split(r",", "a,b,c"))      # ['a', 'b', 'c']
print(re.split(r"(,)", "a,b,c"))    # ['a', ',', 'b', ',', 'c']
print(re.split(r"\d", "a1b2c"))    # ['a', 'b', 'c']
print(re.split(r"\d", "1a2b3"))    # ['', 'a', 'b', '']
```

Splitting at a position

Lookaround (covered in the next chapter) can split without consuming anything. As a preview, this cuts after sentence-ending punctuation but keeps it attached. The second example splits on blank lines.

```
print(re.split(r"(?<=[.!?]) +", "Hi. How are you? Fine!"))
# ['Hi.', 'How are you?', 'Fine!']

print(re.split(r"\n\s*\n", "p1 line\nstill p1\n\np2"))
# ['p1 line\nstill p1', 'p2']
```

NEED	PLAIN STRING METHOD	REGEX
Fixed text replaced	`s.replace("a", "b")`	Not needed
Varying text replaced	not possible	`re.sub(pattern, repl, s)`
Split on one fixed separator	`s.split(",")`	Not needed
Split on any run of spaces	`s.split()`	Not needed
Split on several or variable separators	not possible	`re.split(pattern, s)`

CHOOSING THE TOOL

If the text is fixed and the separator is a single fixed string, the plain string method is simpler and faster. Reach for `re.sub` and `re.split` when the thing you are replacing or splitting on varies.

Challenges

EXERCISE 1

Given `text = "Smith, Ann: 2026-11-02\nJones, Bo: 2025-01-09"`, use one `re.sub` call with named groups and `re.M` to turn each line into `First Last: dd/mm/yyyy`. Then use `re.subn` to count how many lines have a `Last, First` name at the start.

View solution

EXERCISE 2

Use a replacement function to convert every Celsius reading in `"Mon 20C, Tue -5C, Wed 21.5C, 3Cats"` to Fahrenheit with one decimal place (for example `68.0F`). Negative and decimal values must work, and `3Cats` must be left alone. Say how your pattern protects it.

View solution

EXERCISE 3

Split `"a1+b2*(c-3)"` into tokens while keeping the operators and brackets, and filter out the empty strings. Then split `"Para one\nline two\n\n\nPara two\n\nPara three"` into paragraphs on one or

more blank lines, and show the effect of `maxsplit=1`.

View solution

QUICK REFERENCE

- **re.sub(p, r, s, count=0):** returns a new string; `re.subn` also returns the count
- **In the replacement:** `\1`, `\g<1>`, `\g<name>` insert groups
- **In the pattern:** `\1` demands a repeat of group 1
- **Function replacement:** receives the match, must return a string
- **Literal text as a pattern:** `re.escape(text)`
- **re.split(p, s, maxsplit=0):** a capturing group keeps the separators
- **Empty strings:** appear between adjacent separators and at the ends; filter if unwanted

COMING NEXT

Regular Expressions in Practice 5 covers lookaround: lookahead and lookbehind, which test the surroundings of a match without consuming them.

Regular Expressions in Practice

Chapter 5 · Lookaround

Every piece of a pattern so far has either matched characters or failed. **Lookaround** is different: it checks what sits just before or just after the current position, then reports yes or no *without using up any characters*. Because the check has no width, the text it inspects is not part of the match, and the next part of the pattern starts from the same spot.

> **WHAT WAS RUN**
>
> As in the earlier chapters, every snippet was executed and the comments show what Python printed.

SYNTAX	NAME	SUCCEEDS WHEN THE TEXT…
`(?=...)`	Positive lookahead	after this point matches `...`
`(?!...)`	Negative lookahead	after this point does *not* match `...`
`(?<=...)`	Positive lookbehind	before this point matches `...`
`(?<!...)`	Negative lookbehind	before this point does *not* match `...`

Lookahead

A positive lookahead `(?=...)` demands that something follows, but leaves it out of the result.

```
import re

print(re.findall(r"\w+(?=:)", "key: value, other: 2"))
# ['key', 'other']

print(re.findall(r"\d+(?= dollars)", "5 dollars, 7 euros, 12 dollars"))
# ['5', '12']
```

The colon and the word "dollars" were checked but are not in the results, which hold only the key names and the numbers. Compare that with `\w+:`, which would have returned `'key:'`

with the colon attached.

A negative lookahead `(?!...)` demands that something does *not* follow.

```
print(re.findall(r"\bfoo(?!bar)\w*", "foobar foobaz foo"))
# ['foobaz', 'foo']

print(re.findall(r"\bcat\b(?! food)", "cat food, cat toy, cat"))
# ['cat', 'cat']
```

The first pattern reads "foo, as long as it is not immediately followed by bar, then the rest of the word". The second finds every `cat` except the one in `cat food`.

Lookbehind

Lookbehind looks at the text *before* the current position. `(?<=...)` requires it; `(?<!...)` forbids it.

```
print(re.findall(r"(?<=\$)\d+", "cost $30 or 40 or $5"))
# ['30', '5']

print(re.findall(r"(?<!\$)\b\d+", "cost $30 or 40 or $5"))
# ['40']

print(re.findall(r"(?<=<b>).*?(?=</b>)", "<b>one</b> and <b>two</b>"))
# ['one', 'two']
```

In the second pattern the `\b` matters. Without it, the engine could start in the middle of `$30`, at the `0`, where the character before is a `3`, not a dollar sign, and report a stray `0`. The word boundary stops a match from starting in the middle of a number. The third pattern pairs a lookbehind and a lookahead to pull out the text between two tags, leaving the tags out.

Lookbehind must have a fixed width

Python needs to know how far back to step, so the text inside a lookbehind must have a fixed length. Quantifiers such as `+` and `*` are not allowed there.

```
try:
    re.compile(r"(?<=a+)b")
except re.error as e:
    print("error:", e)
# error: look-behind requires fixed-width pattern

print(re.findall(r"(?<=ab)c|(?<=x)c", "abc xc zc"))
# ['c', 'c']
```

Alternatives of *different* widths inside one lookbehind are rejected too. The workaround is the one on the last line: write a separate lookbehind for each width and join them with `|` outside. (Third-party modules such as `regex` lift this restriction; the standard `re` module does not.)

Zero-width means nothing is consumed

Because a lookahead does not move the position, it can be used to find *overlapping* matches, which a normal pattern cannot, since each match uses up its characters. The trick is to put the whole pattern inside a lookahead and capture it.

```
print(re.findall(r"\w\w", "abcd"))          # ['ab', 'cd']
print(re.findall(r"(?=(\w\w))", "abcd"))    # ['ab', 'bc', 'cd']
print(re.findall(r"(?=(aa))", "aaaa"))     # ['aa', 'aa', 'aa']
```

At each position the lookahead checks whether two word characters follow and captures them; nothing is consumed, so the engine steps forward one character at a time and every position gets its chance.

Lookaround in practice

Matching a position, not text

A pattern made only of lookarounds matches an empty string at a *position*. Combined with `re.sub` and `re.split` from the last chapter, that lets you insert or cut at a place without deleting anything.

```
# Insert thousands separators: a position that follows a digit and
# is followed by a whole number of three-digit groups up to a boundary.
t = "Sales: 1234567 units, 89 units, 1000000 units, 999 units"
print(re.sub(r"(?<=\d)(?=(?:\d{3})+\b)", ",", t))
# Sales: 1,234,567 units, 89 units, 1,000,000 units, 999 units

# camelCase to snake_case: insert "_" between a lowercase letter/digit
# and an uppercase letter.
for w in ["camelCaseWordsHere", "myVar2Name", "already_snake"]:
    print(w, "->", re.sub(r"(?<=[a-z0-9])(?=[A-Z])", "_", w).lower())
# camelCaseWordsHere -> camel_case_words_here
# myVar2Name -> my_var2_name
# already_snake -> already_snake

# Split after each comma but keep the comma with its piece.
print(re.split(r"(?<=,)", "a,b,c"))
# ['a,', 'b,', 'c']
```

In the thousands example the number `89` and `999` are untouched: from the position inside them, no whole number of three-digit groups reaches the end of the number. (This simple version assumes whole numbers; a decimal such as `1234.5678` would be mishandled, a reminder that every pattern has assumptions.)

Several conditions at once

Several lookaheads placed at the start of a pattern all test from the *same* position, so together they act as a checklist of independent requirements. This is the standard way to express "contains X and Y and not Z".

```
pw = r"^(?=.*\d)(?=.*[A-Z])(?!.*\s).{8,12}$"

for s in ["Abcdefg1", "abcdefg1", "Abcdefg 1", "Abcdefghijkl1", "Ab1"]:
```

```
    print(repr(s), bool(re.fullmatch(pw, s)))
# 'Abcdefg1' True
# 'abcdefg1' False        (no capital)
# 'Abcdefg 1' False       (contains a space)
# 'Abcdefghijkl1' False   (13 characters, too long)
# 'Ab1' False             (too short)
```

Each `(?=.*X)` means "somewhere ahead there is an X". They do not consume anything, so the final `.{8,12}` still gets to match the whole string.

Skipping lines

```
text = "# note\ncode\n  # x\nmore"
print(re.findall(r"^(?!#).+", text, flags=re.M))
# ['code', '  # x', 'more']
```

The pattern keeps lines that do not *start* with `#`. Note that `' # x'` survives, because its first character is a space; a pattern is only as clever as what you wrote.

> **WHEN LOOKAROUND IS THE WRONG TOOL**
>
> If you only need the part you captured, a normal capturing group is simpler: `re.findall(r"(\w+):", s)` returns the same keys as `\w+(?=:)`. Reach for lookaround when the surrounding text must not be consumed (overlaps, substitutions at a position) or when you need several independent conditions on the same text.

Challenges

EXERCISE 1

Given `"Sales: 1234567 units, 89 units, 1000000 units, 999 units"`, use `re.sub` with only lookarounds in the pattern to insert commas. Then explain, in a sentence, why `89` and `999` are not changed.

View solution

EXERCISE 2

Write one pattern, using lookaheads, that accepts a string only if it is 6 to 10 characters long, contains at least one lowercase letter and at least one digit, and contains no spaces. Test it on `"abc123"`, `"ABC123"`, `"abcdef"`, `"abc 123"` and `"a1"`.

View solution

EXERCISE 3

From `"a [one] b [two words] c [] d"`, extract the text inside square brackets without the brackets, using lookbehind and lookahead. Make the first version return the empty brackets as an empty string, then change it so that empty brackets are ignored.

View solution

QUICK REFERENCE

- **(?=...) / (?!...):** what follows must / must not match; nothing is consumed
- **(?<=...) / (?<!...):** what precedes must / must not match; the inside needs a fixed width
- **Different widths:** use separate lookbehinds joined with `|` outside them
- **Overlapping matches:** `(?=(pattern))` with `findall`
- **Position-only patterns:** combine lookarounds with `re.sub` or `re.split` to insert or cut without deleting
- **Checklists:** several `(?=...)` at the start all test from the same position

COMING NEXT

Regular Expressions in Practice 6 covers compiling patterns with `re.compile` and walking through matches one at a time with `finditer`.

Regular Expressions in Practice

Chapter 6 · Compiling and Iterating

So far each call has handed `re` a pattern string and let it do the work. This chapter covers two tools that make larger jobs tidier: **compiled patterns**, which let you name a pattern once and reuse it, and **iteration**, which lets you walk through matches one at a time and see exactly where each one sits. Together they make it possible to write a small *scanner*: a loop that reads text from left to right and labels each piece.

> **WHAT WAS RUN**
>
> As in the earlier chapters, every snippet was executed and the comments show what Python printed.

Compiling a pattern

`re.compile(pattern, flags)` returns a **pattern object**. It has the same operations as the module (`search`, `match`, `fullmatch`, `findall`, `finditer`, `sub`, `split`), but the pattern and flags are already fixed, so you do not pass them again.

```
import re

pat = re.compile(r"\d{4}-\d{2}-\d{2}")

print(pat.findall("on 2026-11-02 and 2027-01-15"))
# ['2026-11-02', '2027-01-15']

print(pat.search("due 2026-11-02"))
# <re.Match object; span=(4, 14), match='2026-11-02'>

print(pat.sub("<date>", "on 2026-11-02 and 2027-01-15"))
# on <date> and <date>

print(pat.pattern)
# \d{4}-\d{2}-\d{2}
```

Why bother?

- **A name for the idea.** `date_re` or `order_re` says what the pattern is for, and one definition near the top of a file is easier to maintain than the same string pasted in three places.
- **Flags travel with it.** `re.compile(r"...", re.I | re.X)` is written once; later calls need no `flags=`.
- **Speed, a little.** The module functions compile the pattern behind the scenes and keep a small internal cache, so calling `re.search` repeatedly is not wildly slow. Compiling yourself skips that lookup and avoids any chance of recompiling when the cache is busy. Treat speed as a minor benefit and clarity as the main one.

A pattern object also reports facts about itself, and compiling an already-compiled pattern simply returns the same object.

```
p2 = re.compile(r"(?P<h>\d+):(?P<m>\d+)", re.I)
print(p2.groups, p2.groupindex)
# 2 {'h': 1, 'm': 2}

words = re.compile(r"[a-z]+", re.I)
print(words.findall("Hello, World"))
# ['Hello', 'World']
print(re.compile(words) is words)
# True
```

Where to put the compile

Compile at module level (or once before a loop), not inside a function that is called many times. The compiled object is plain data and can be shared freely.

Match objects and positions

`findall` returns strings, which is convenient but throws away *where* each match was. A match object keeps that. `m.start()`, `m.end()` and `m.span()` give positions in the original string, and `m.group(n)` or `m["name"]` gives the text.

finditer

`finditer` does what `findall` does, but yields a *match object* for each non-overlapping match, lazily, one at a time.

```
text = "Order 66 shipped; order 7 pending; ORDER 105 done"

for m in re.finditer(r"order (\d+)", text, re.I):
    print(m.group(1), m.start(), m.end(), m.span())
# 66 0 8 (0, 8)
# 7 18 25 (18, 25)
# 105 35 44 (35, 44)

print(re.findall(r"order (\d+)", text, re.I))
# ['66', '7', '105']
```

Use `finditer` when you need positions (to highlight, to slice the original text, to report line and column), when you want several groups by name, or when the text is large and you would rather not build one big list. Use `findall` when the strings are all you need.

```
pairs = re.compile(r"(?P<key>\w+)=(?P<val>\w+)")

for m in pairs.finditer("a=1 b=22 c=x"):
    print(m.groupdict(), m.span("val"))
# {'key': 'a', 'val': '1'} (2, 3)
# {'key': 'b', 'val': '22'} (6, 8)
# {'key': 'c', 'val': 'x'} (11, 12)

d = {m["key"]: m["val"] for m in pairs.finditer("a=1 b=22 c=x")}
print(d)
# {'a': '1', 'b': '22', 'c': 'x'}
```

The iterator is single-use: once it has been walked, create a new one to go again. Empty matches are reported too, and the engine steps past them so the loop always ends:

```
print([m.span() for m in re.finditer(r"\d*", "a12")])
# [(0, 0), (1, 3), (3, 3)]
```

Starting part-way: pos and endpos

A compiled pattern's `search`, `match`, `fullmatch`, `findall` and `finditer` accept optional `pos` and `endpos` arguments that restrict where the engine looks, without slicing the string. Positions in the results still refer to the *whole* string.

```
p = re.compile(r"\d+")
s = "ab12cd345"

print(p.search(s, 3))
# <re.Match object; span=(3, 4), match='2'>
print(p.match(s, 2))
# <re.Match object; span=(2, 4), match='12'>
print(p.match(s, 0))
# None
print(p.findall(s, 0, 4))
# ['12']
```

Two details are worth remembering. First, `p.search(s, 3)` starts looking *at* index 3, which is in the middle of `12`, so it finds only `'2'`. Second, `match(s, pos)` insists that the match begin exactly at `pos`, which is the key to the scanner loop below. A caution: `^` does not match at `pos` unless `pos` is really the start of the string (or of a line, with `re.M`):

```
print(re.compile(r"^\d+").search(s, 2))
# None
```

So when scanning with `pos`, leave `^` out and let `match` do the anchoring.

A scanner with named groups

A **tokenizer** splits text into labelled pieces: numbers, names, operators. With regex the recipe is one alternation of named groups, one group per kind of token. The match object's `lastgroup` attribute gives the name of the group that matched, so you never need to test which group is not `None`.

```
TOKEN = re.compile(r"""
    (?P<NUM>\d+(?:\.\d+)?)
  | (?P<NAME>[A-Za-z_]\w*)
  | (?P<OP>[-+*/=()])
  | (?P<SKIP>\s+)
  | (?P<BAD>.)
""", re.VERBOSE)

def tokens(src):
    for m in TOKEN.finditer(src):
        kind = m.lastgroup
        if kind == "SKIP":
            continue
        if kind == "BAD":
            raise ValueError(f"unexpected {m.group()!r} at {m.start()}")
        yield kind, m.group()

print(list(tokens("rate = (3.5 + x1) * 2")))
# [('NAME', 'rate'), ('OP', '='), ('OP', '('), ('NUM', '3.5'), ('OP', '+'),
#  ('NAME', 'x1'), ('OP', ')'), ('OP', '*'), ('NUM', '2')]

try:
    list(tokens("a $ b"))
except ValueError as e:
    print("error:", e)
# error: unexpected '$' at 2
```

Several design points are packed into that example:

- **Order matters.** Alternatives are tried left to right, so put more specific kinds first. The final `BAD` group is a catch-all, so anything unrecognised surfaces as an error with a position instead of being silently skipped.

- **Full coverage.** Because `SKIP` and `BAD` together cover every character, the matches tile the whole input with no gaps. Without a catch-all, `finditer` would skip unrecognised characters without a word.
- **It is a generator.** `yield` hands tokens out lazily, so a caller can stop early.

A manual loop with match(string, pos)

`finditer` is allowed to skip over text that nothing matches. When you need the opposite, a hard guarantee that every character is accounted for, move a position forward yourself and use `match` at that position. If it fails, you know exactly where the input went wrong.

```
NUM = re.compile(r"\d+")
WS  = re.compile(r"\s*")

def parse_ints(s):
    out, pos = [], 0
    while pos < len(s):
        pos = WS.match(s, pos).end()
        if pos >= len(s):
            break
        m = NUM.match(s, pos)
        if not m:
            raise ValueError(f"bad input at {pos}: {s[pos]!r}")
        out.append(int(m.group()))
        pos = WS.match(s, m.end()).end()
        if pos < len(s):
            if s[pos] != ",":
                raise ValueError(f"expected comma at {pos}")
            pos += 1
    return out

print(parse_ints("1, 22 ,333"))
# [1, 22, 333]

try:
    parse_ints("1, x")
except ValueError as e:
    print("error:", e)
```

```
# error: bad input at 3: 'x'
```

Here `WS` can match an empty string, so `WS.match(s, pos)` always succeeds and simply returns the position after any spaces. The loop is the scanner: each pass consumes one piece, and `pos` only ever moves forward.

NEED	REACH FOR
Just the matched strings	`findall`
Positions, named groups, or lazy processing	`finditer`
Labelled pieces covering all the text	Named groups with a catch-all, plus `lastgroup`
A guarantee about every character, with exact error positions	A `while` loop using `match(s, pos)`
Same pattern used in several places	`re.compile` once, near the top

LASTGROUP AND LASTINDEX

`m.lastgroup` is the name of the last matched capturing group, and `m.lastindex` is its number. With a top-level alternation of groups, that is the group that matched. If the matching group has no name, `lastgroup` is `None` but `lastindex` still works. Nested groups can make "last" surprising, so keep scanner groups flat and use `(?:...)` for any grouping inside them, as `NUM` does above.

Challenges

EXERCISE 1

Given `"GET /a 200 12ms; POST /b 404 8ms; GET /c 200 130ms"`, compile one pattern with named groups `method`, `path`, `status` and `ms`. Use `finditer` to print each method, path and the position where its record starts, then print the total of the `ms` values.

View solution

EXERCISE 2

Write a scanner for `"go 12 steps, then 7 more!"` with named groups `INT`, `WORD`, `SP` (spaces) and a catch-all `X` for anything else. Print the list of `(kind, text)` pairs without the spaces, and use `collections.Counter` on `lastgroup` to count how many tokens of each kind there were.

View solution

EXERCISE 3

Split `say 'hello world' to "big wide" world` into words, keeping a quoted phrase together (single or double quotes) and removing the quote marks. Use one compiled pattern with an alternation and `finditer`; expected result: `['say', 'hello world', 'to', 'big wide', 'world']`. Hint: a match has several groups, only one of which is not `None`.

View solution

QUICK REFERENCE

- **re.compile(p, flags):** returns a pattern object with the same methods as the module; compile once, reuse, and give it a meaningful name
- **finditer:** lazy iterator of match objects; use `start()`, `end()`, `span()`, `group()`, `groupdict()`
- **findall vs finditer:** strings only versus full match objects with positions
- **pos / endpos:** limit the search on a compiled pattern; results still index the whole string; `^` ignores `pos`
- **match(s, pos):** must match exactly at `pos`; the basis of a manual scanner loop
- **Tokenizer recipe:** flat named groups in one alternation, a catch-all last, and `m.lastgroup` to label each piece

COMING NEXT

Regular Expressions in Practice 7 applies everything so far to validating real-world data such as dates, phone numbers and email addresses, and looks honestly at where regular expressions stop being the right tool.

Regular Expressions in Practice

Chapter 7 · Validating Real Data

Validation is the job most people first reach for regular expressions to do: is this text a date, a phone number, an email address? The tools from the earlier chapters are enough, but there is a skill to using them well. This chapter shows the habits that make validators reliable, and is honest about the point where a pattern should hand over to ordinary code.

> **WHAT WAS RUN**
>
> As in the earlier chapters, every snippet was executed and the comments show what Python printed.

Shape first, meaning second

A pattern is very good at checking **shape**: how many characters, of what kind, in what order. It is poor at checking **meaning**: whether month 13 exists, or whether 30 February is a real day. The reliable approach is to split the work. Let the pattern check the shape and capture the parts, then let normal Python check the meaning.

```
import re, datetime

date_re = re.compile(r"(\d{4})-(\d{2})-(\d{2})")

def valid_date(s):
    m = date_re.fullmatch(s)
    if not m:
        return False
    y, mo, d = map(int, m.groups())
    try:
        datetime.date(y, mo, d)
    except ValueError:
        return False
    return True

for s in ["2026-11-02", "2026-13-01", "2026-02-30", "2024-02-29", "26-1-2"]:
    print(repr(s), valid_date(s))
# '2026-11-02' True
```

```
# '2026-13-01' False
# '2026-02-30' False
# '2024-02-29' True
# '26-1-2' False
```

A single pattern that also knew about month lengths and leap years would be long and fragile. The standard library already knows the calendar, so the pattern's only task was to get the three numbers out safely.

Use fullmatch, not match with $

A validator must say yes only when the *entire* string is acceptable. There is a trap here: `$` matches at the very end of the string *and also just before a final newline*. Input read from files or forms often has a stray newline, and it slips through:

```
print(re.match(r"\d+$", "42\n"))
# <re.Match object; span=(0, 2), match='42'>
print(re.fullmatch(r"\d+", "42\n"))
# None
```

`fullmatch` has no such gap, and it also removes the need to remember both anchors. Make it the default for validation. (The alternative is `\Z`, which matches only at the true end.)

Beware: \d accepts more than 0-9

For text (`str`) patterns, `\d` matches any Unicode decimal digit, not only the ASCII ones. If you mean exactly `0-9`, say so, or use the `re.ASCII` flag.

```
n = "٣٤"   # two Arabic-Indic digits
print(bool(re.fullmatch(r"\d+", n)),
      bool(re.fullmatch(r"[0-9]+", n)),
      bool(re.fullmatch(r"\d+", n, re.A)))
```

```
# True False False
```

Python's `int()` happily converts such digits, so this is not always a problem, but a validator whose job is "ASCII digits only" should be explicit.

Phone numbers: accept messy input, store clean output

People type phone numbers with spaces, hyphens and different prefixes. A good validator tolerates the common variations and then **normalises** to one form. Here the pattern uses `re.VERBOSE` so each part is labelled, and captures just the digits it cares about.

```
phone_re = re.compile(r"""
    (?:\+44\s?|0)       # country code or trunk zero
    (\d{4})             # area
    [\s-]?
    (\d{6})
""", re.X)

def norm_phone(s):
    m = phone_re.fullmatch(s.strip())
    return "+44" + m[1] + m[2] if m else None

for s in ["+44 1314 960123", "01314-960123", "01314 96012", "(0131) 496 0123"]:
    print(repr(s), norm_phone(s))
# '+44 1314 960123' +441314960123
# '01314-960123' +441314960123
# '01314 96012' None
# '(0131) 496 0123' None
```

The last input is a perfectly real way to write a number, but this pattern does not allow brackets. That is a design decision: you either extend the pattern for every format you meet, or you clean the input first and validate the cleaned version. Stripping everything except digits is a one-liner:

```
print(re.sub(r"\D", "", "(0131) 496 0123"))
```

```
# 01314960123
```

Cleaning first and checking the digit count afterwards is often simpler than one heroic pattern that allows every punctuation style.

Email addresses: the honest version

The full rules for email addresses are enormously complicated, and a pattern that follows them exactly is famously huge. In practice you want a *sanity check* that catches obvious typos, not a proof that the mailbox exists. A modest pattern does that:

```
email_re = re.compile(r"[\w.+-]+@[A-Za-z0-9-]+(?:\.[A-Za-z0-9-]+)+")

for s in ["ann@example.com", "a.b+tag@mail.co.uk", "ann@@example.com",
          "ann@example", "@example.com", "ann@exa mple.com", "a..b@example.com"]:
    print(s, bool(email_re.fullmatch(s)))
# ann@example.com True
# a.b+tag@mail.co.uk True
# ann@@example.com False
# ann@example False
# @example.com False
# ann@exa mple.com False
# a..b@example.com True
```

It rejects the typical slips (double `@`, no domain dot, empty name, a space). It also accepts `a..b@example.com`, which strictly is not valid. We could tighten the pattern to forbid doubled dots, but each tightening makes it longer and still leaves gaps in the other direction. This is the central point of the chapter:

> **THE LIMIT OF VALIDATION**
>
> A pattern can tell you an email address *looks* plausible. Only sending a message to it and getting a response proves it works. Use a loose pattern to catch typos early, then confirm by the real route (a confirmation message), rather than growing the pattern.

Numbers with a range: split the job

The same shape-then-meaning idea applies to anything with a numeric range. An IPv4 address is four numbers from 0 to 255. Writing "0 to 255" as a pattern is possible but ugly; checking `int(g) <= 255` is clear.

```
ip_re = re.compile(r"(\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})")

def valid_ip(s):
    m = ip_re.fullmatch(s)
    return bool(m) and all(int(g) <= 255 for g in m.groups())

for s in ["192.168.1.1", "256.1.1.1", "1.2.3", "10.0.0.255"]:
    print(s, valid_ip(s))
# 192.168.1.1 True
# 256.1.1.1 False
# 1.2.3 False
# 10.0.0.255 True
```

Where the shape is genuinely tidy, a pattern alone is fine. A hex colour is either three or six hex digits after a `#`:

```
hex_re = re.compile(r"#(?:[0-9a-f]{3}|[0-9a-f]{6})", re.I)

for s in ["#fff", "#A1B2C3", "#ffff", "fff"]:
    print(s, bool(hex_re.fullmatch(s)))
# #fff True
# #A1B2C3 True
# #ffff False
# fff False
```

CHECK	PATTERN ALONE?	BETTER APPROACH
Hex colour, simple codes	Yes	`fullmatch` with a clear pattern
Calendar date	Shape only	Pattern for parts, `datetime.date` for meaning

CHECK	PATTERN ALONE?	BETTER APPROACH
Number in a range (time, IPv4)	Shape only	Pattern for parts, `int()` comparisons
Phone number	Shape, per format	Clean to digits, then check length and prefix
Email address	Plausibility only	Loose pattern plus a confirmation message

Checklist for a trustworthy validator

- Use `fullmatch`, so the whole input must fit and a trailing newline is rejected.
- Be explicit about character sets: `[0-9]` or `re.A` when you mean ASCII digits.
- Capture the parts, then check ranges and rules in code.
- Normalise on the way in so the rest of the program sees one format.
- Test the failures as carefully as the successes: write the bad inputs down and run them.
- Decide what "good enough" is. A validator that is slightly loose and understandable beats one that is exact and unreadable.

Challenges

EXERCISE 1

Write `valid_time(s)` for 24-hour times in the form `HH:MM`. Use a pattern for the shape and ordinary comparisons for the ranges (hours up to 23, minutes up to 59). Test it on `09:30`, `23:59`, `24:00`, `12:60` and `9:30`.

View solution

EXERCISE 2

Write one compiled pattern that checks the shape of a postcode: one or two capital letters, a digit, an optional capital letter or digit, a single space, a digit, then two capital letters. Test it on `EH1 1YZ`, `SW1A 1AA`, `M1 1AE`, `eh1 1yz`, `EH11 1YZ` and `EH1 1Y` and print True or False for each.

View solution

EXERCISE 3

A username must start with a letter, then have 2 to 15 more letters, digits or underscores. Write `check(s)` returning `"ok"` or `"invalid"` using `fullmatch`. Show that `"ok\n"` is rejected, and show that `re.match(r"[A-Za-z]\w{2,15}$", "bob\n")` wrongly accepts a trailing newline.

View solution

QUICK REFERENCE

- **fullmatch:** the validation default; whole string must match, trailing newline rejected
- **$ gap:** `$` also matches before a final newline, so `match(...$)` can accept `"42\n"`
- **\d:** any Unicode digit for `str`; use `[0-9]` or `re.A` for ASCII only
- **Shape then meaning:** capture parts with groups, then check ranges and calendars in Python
- **Normalise:** strip or reformat input so later code sees one format
- **Email:** a loose sanity check only; confirm by really contacting the address

COMING NEXT

Regular Expressions in Practice 8 turns to parsing logs: pulling fields out of lines of text and building a small report from them.

Regular Expressions in Practice

Chapter 8 · Parsing Logs

Log files are where patterns earn their keep. Each line is a small record written by a program, so the lines share a layout, but the layout is plain text and there is no ready-made parser. In this chapter you will turn raw lines into dictionaries, cope with lines that do not fit, and finish with a short report. It pulls together named groups, `re.VERBOSE`, `fullmatch` and the iteration tools from the earlier chapters.

> **WHAT WAS RUN**
>
> As before, every snippet was executed and the comments show what Python printed.

Read the line first

Start by looking at the structure, not by writing a pattern. A line in our example log looks like this:

```
2026-11-02 09:15:03 ERROR [auth] Login failed for user=ann ip=10.0.0.5
```

There are five parts: a date, a time, a level in capitals, a source in square brackets, and then a free-text message that may hold `key=value` pairs. The first four parts are rigid; the message is loose. That suggests two steps: one pattern for the fixed frame, and a second, simpler one for the pairs inside the message.

The frame pattern

Named groups make the result self-describing, and `re.VERBOSE` lets each part carry a label. Note that in verbose mode the literal spaces are written as `\s`, because plain spaces in the pattern are ignored.

```
import re

log_re = re.compile(r"""
    (?P<date>\d{4}-\d{2}-\d{2})\s
    (?P<time>\d{2}:\d{2}:\d{2})\s
    (?P<level>[A-Z]+)\s
    \[(?P<source>\w+)\]\s
    (?P<msg>.*)
""", re.X)

line = "2026-11-02 09:15:03 ERROR [auth] Login failed for user=ann ip=10.0.0.5"
m = log_re.fullmatch(line)
print(m.groupdict())
# {'date': '2026-11-02', 'time': '09:15:03', 'level': 'ERROR', 'source': 'auth', 'msg':
```

`groupdict()` returns exactly the dictionary we wanted. The last group uses `.*` deliberately: it is the one place where "anything" is the right answer, and because it is at the end and the match is anchored by `fullmatch`, it cannot swallow the earlier fields.

The key=value pattern

For the message, `findall` with two groups returns a list of pairs, and `dict()` turns that list straight into a dictionary:

```
kv_re = re.compile(r"(\w+)=(\S+)")

print(dict(kv_re.findall(m["msg"])))
# {'user': 'ann', 'ip': '10.0.0.5'}
print(kv_re.findall("no pairs here"))
# []
```

A message with no pairs simply gives an empty list, and so an empty dictionary. No special case is needed. This is the benefit of using `findall` rather than `search` here: "nothing found" is a normal, harmless result.

Never assume every line is good

Real logs contain blank lines, truncated lines and lines from some other program. A parser that raises an error on the first odd line is useless on a big file. The habit is to return `None` for a line that does not match, and let the caller decide what to do. Here is the complete parser, with the bad lines collected rather than hidden:

```
def parse_line(line):
    m = log_re.fullmatch(line)
    if not m:
        return None
    rec = m.groupdict()
    rec["fields"] = dict(kv_re.findall(rec.pop("msg")))
    return rec

def parse_log(text):
    records, bad = [], []
    for n, line in enumerate(text.splitlines(), 1):
        rec = parse_line(line)
        if rec:
            records.append(rec)
        else:
            bad.append((n, line))
    return records, bad

logs = """\
2026-11-02 09:15:03 INFO [web] Request ok path=/home ms=12
2026-11-02 09:15:07 ERROR [auth] Login failed for user=ann ip=10.0.0.5
this line is damaged
2026-11-02 09:16:41 WARN [web] Slow request path=/search ms=870
2026-11-02 09:17:02 ERROR [db] Timeout after ms=3000
2026-11-02 09:18:30 ERROR [auth] Login failed for user=bob ip=10.0.0.9
2026-11-02 09:19:12 INFO [web] Request ok path=/home ms=15
"""

records, bad = parse_log(logs)
print(len(records), "parsed,", len(bad), "skipped")
for n, line in bad:
    print("  line", n, "skipped:", line)
# 6 parsed, 1 skipped
```

```
#   line 3 skipped: this line is damaged
```

Two details are worth noticing. `splitlines()` removes the line endings, so the pattern never has to think about `\n`. And `rec.pop("msg")` removes the raw message as it is used, so each record holds the structured fields only.

From records to a report

Once each line is a dictionary, the regular expressions are finished. Everything else is ordinary Python, and `collections.Counter` does most of the counting:

```
from collections import Counter

print("By level:")
for lvl, c in Counter(r["level"] for r in records).most_common():
    print(f"  {lvl:<6}{c}")

errors = Counter(r["source"] for r in records if r["level"] == "ERROR")
print("Errors by source:", dict(errors))

timed = [r for r in records if "ms" in r["fields"]]
slowest = max(timed, key=lambda r: int(r["fields"]["ms"]))
print("Slowest:", slowest["time"], slowest["source"], slowest["fields"]["ms"] + " ms")

ips = sorted({r["fields"]["ip"] for r in records if "ip" in r["fields"]})
print("IPs:", ips)
# By level:
#   ERROR 3
#   INFO  2
#   WARN  1
# Errors by source: {'auth': 2, 'db': 1}
# Slowest: 09:17:02 db 3000 ms
# IPs: ['10.0.0.5', '10.0.0.9']
```

Values from `kv_re` are always strings, so numbers such as `ms` are converted with `int()` at the point of use. Converting too early, before you know a field is present and numeric, is a

common source of errors.

JOB	TOOL	WHY
Split a line into fixed fields	One `fullmatch` with named groups	The whole line must fit; fields come back by name
Pull out a variable number of pairs	`findall` then `dict()`	Zero pairs is a normal, empty result
Skip bad lines safely	Return `None`, collect the failures	One odd line must not stop the run
Count and summarise	`Counter`, `max`, `sorted`	No pattern needed once the data is structured

Practical habits

- Look at real lines first and note which parts are rigid and which are loose.
- Use two simple patterns (a frame and the details) rather than one that does everything.
- Use named groups so the code reads like the log format.
- Keep the bad lines and report how many there were. Silent skipping hides format changes.
- Parse once into dictionaries, then do all counting and filtering on the structured data.
- Large files: loop over the open file object line by line rather than reading everything into one string.

Challenges

EXERCISE 1

Write `parse_access(line)` for web-server style lines such as `192.168.1.5 - - [02/Nov/2026:09:15:03] "GET /index.html HTTP/1.1" 200 512`. Use named groups and return a dictionary with `ip`, `when`, `method`, `path`, `status` (as an int) and `size` (as an int, with `-` meaning 0), or `None` if the line does not match. Test it on that line, on `10.0.0.8 - - [02/Nov/2026:09:15:09] "POST /login HTTP/1.1" 302 -`, and on `not an access line`.

View solution

EXERCISE 2

Using the `parse_log` function from this chapter, parse a six-line log with these levels in order: INFO, WARN, INFO, ERROR, WARN, INFO (use any sources and messages in the same line format). Print each level with its count, sorted by count from highest to lowest, with ties in alphabetical order.

View solution

EXERCISE 3

Logs must not leak addresses. Given the lines `10:00 user=ann ip=10.0.0.5 action=login`, `10:01 user=bob ip=192.168.1.20 action=logout` and `10:02 action=ping`, use `re.sub` with a backreference to replace the address after `ip=` with `x.x.x.x` while keeping `ip=` itself. The line with no address must come through unchanged.

View solution

QUICK REFERENCE

- **Frame pattern:** `re.X` with named groups and `\s` for spaces, matched with `fullmatch`
- **groupdict():** the match as a dictionary keyed by group name
- **findall + dict:** turns `key=value` pairs into a dictionary; no pairs gives `{}`
- **Bad lines:** return `None` and collect them with their line numbers
- **splitlines:** strips line endings so patterns ignore `\n`
- **Report:** `Counter` and `max` on the parsed records; convert numbers with `int()` when used

COMING NEXT

Regular Expressions in Practice 9 looks at performance and pitfalls: how a pattern can become slow, what causes it, and when not to use a pattern at all.

Regular Expressions in Practice

Chapter 9 · Performance and Pitfalls

Most patterns are fast and most mistakes are quiet: the pattern runs, returns something, and the something is wrong. This chapter collects the problems that cause the most trouble in practice. First comes the rare but severe one, a pattern that takes minutes to fail. Then come the everyday traps, and finally the question of when a plain string method is the better tool.

> **WHAT WAS RUN**
>
> Every snippet was executed and the comments show what Python printed. Timings differ from machine to machine, so the examples compare timings with a test rather than printing raw numbers.

Backtracking in one paragraph

When a pattern has a choice to make (a quantifier that can take more or fewer characters, or an alternation), the engine picks one option and remembers the others. If the rest of the pattern fails, it goes back and tries the next option. That is *backtracking*, and it is what makes patterns flexible. It is harmless when there are few choices. It becomes a problem when the number of ways to split the text grows very fast.

Catastrophic backtracking

The classic cause is a quantifier inside a quantifier, where both can match the same characters. In `^(a+)+$` the inner `a+` and the outer `+` can share a run of letters in many different ways. If the text then fails at the very end (a trailing `b`), the engine tries every one of those ways before giving up. Each extra character roughly doubles the work.

```
import re, time

def took(pattern, text):
    start = time.perf_counter()
    pattern.search(text)
    return time.perf_counter() - start
```

```
risky = re.compile(r"^(a+)+$")
t18 = took(risky, "a" * 18 + "b")
t22 = took(risky, "a" * 22 + "b")
print("four more characters, at least 5x slower:", t22 > 5 * t18)

safe = re.compile(r"^a+$")
print("safe pattern, 100000 characters, under 0.1s:", took(safe, "a" * 100000 + "b") < 0
# four more characters, at least 5x slower: True
# safe pattern, 100000 characters, under 0.1s: True
```

Four extra characters made the risky pattern many times slower, and the growth continues: a few dozen characters would take longer than you would wait. The safe pattern handles a hundred thousand characters with no trouble. Both patterns accept exactly the same strings. Only the amount of work differs.

This matters most when the *text* comes from someone else, such as a web form. A user who submits a long string that nearly matches can tie up your program. This is sometimes called a regular expression denial of service.

How to avoid it

- Do not put a repeat around something that already repeats, unless the pieces cannot match the same text. `(a+)+` is the warning sign.
- Be careful with alternations whose branches overlap inside a repeat, such as `(a|aa)+`.
- Prefer a specific class over `.*`. A pattern that can only match the right characters has fewer choices to explore.
- Test with a long near-miss, not only with matching text. The slow case is the one that fails at the end.
- Limit the length of untrusted input before matching it.

```
# Removing the inner repeat gives the same meaning with no nested choice
print(re.search(r"^(?:a+)$", "a" * 22 + "b"))
print(re.search(r"^(a|aa)+$", "a" * 3) is not None)
# None
# True
```

Pitfall: greedy matching across a boundary

Chapter 1 introduced greedy and lazy quantifiers. It is worth repeating because it is the most common wrong answer. A greedy `.*` runs to the *last* possible end, so it can swallow several items at once. There are two fixes, and the second is usually better:

```
html = "<b>one</b> and <b>two</b>"
print(re.findall(r"<b>.*</b>", html))
print(re.findall(r"<b>.*?</b>", html))
print(re.findall(r"<b>([^<]*)</b>", html))
# ['<b>one</b> and <b>two</b>']
# ['<b>one</b>', '<b>two</b>']
# ['one', 'two']
```

The lazy form stops at the first closing tag. The negated class `[^<]*` says what the text is *allowed* to contain, and so it also gives the engine nothing to backtrack over. A class that excludes the closing delimiter is both clearer and faster.

Pitfall: match is not search

`re.match` only tries the *start* of the string. Reaching for it when you meant `search` gives `None` and no error message.

```
print(re.match(r"\d+", "ab12"))
print(re.search(r"\d+", "ab12").group())
# None
# 12
```

Pitfall: the dollar sign and the newline

`$` matches at the end of the string *and* just before a final newline. So a value that looks validated can still carry a newline. For validation, use `fullmatch`, or `\Z`, which matches only at

the true end.

```
print(re.search(r"^\d+$", "42\n"))
print(re.fullmatch(r"\d+", "42\n"))
print(re.search(r"^\d+\Z", "42\n"))
# <re.Match object; span=(0, 2), match='42'>
# None
# None
```

Pitfall: \d is wider than 0 to 9

For text (`str`) patterns, `\d` matches digits from other writing systems too, which may not be what a validator wants. Use `[0-9]` or the `re.ASCII` flag when you mean the ordinary digits.

```
arabic = "\u0664\u0662"
print(bool(re.fullmatch(r"\d+", arabic)))
print(bool(re.fullmatch(r"[0-9]+", arabic)))
print(bool(re.fullmatch(r"\d+", arabic, re.A)))
# True
# False
# False
```

Pitfall: alternation takes the first branch that works

Alternation is tried left to right and the first branch that lets the whole pattern succeed wins, even if a later branch would match more. Put longer alternatives first.

```
print(re.match(r"cat|category", "category").group())
print(re.match(r"category|cat", "category").group())
# cat
# category
```

Pitfall: user text inside a pattern

If a pattern is built from text you did not write, characters such as `.`, `+` and `(` will be read as pattern syntax. `re.escape` turns them into literals. Without it, the search can match the wrong text, or fail to compile.

```
user = "a.b"
print(re.findall(user, "a.b axb"))
print(re.findall(re.escape(user), "a.b axb"))
print(re.escape("1.5+2"))
# ['a.b', 'axb']
# ['a.b']
# 1\.5\+2
```

When not to use a pattern

A pattern is a tool for text with *shape*. If you are looking for fixed text, the built-in string methods are shorter, faster and cannot be broken by special characters.

```
name = "report_2026.txt"
print(name.endswith(".txt"), name.startswith("report"), "2026" in name)
print(name.replace("_", "-"), name.split("."))
# True True True
# report-2026.txt ['report_2026', 'txt']
```

SITUATION	BETTER TOOL	WHY
Fixed text present or absent	`in`, `startswith`, `endswith`	No syntax to get wrong
Replace one fixed piece of text	`str.replace`	Clear and fast
Split on one fixed separator	`str.split`	No pattern needed

SITUATION	BETTER TOOL	WHY
Text with variable shape (digits, optional parts)	`re`	This is what patterns are for
Nested or recursive structure (HTML, JSON, source code)	A real parser	A pattern cannot track nesting reliably
Untrusted long input	A length limit, then a simple pattern	Bounds the worst case

Practical habits

- Test with long near-misses, not only with matching text.
- Avoid a repeat of something that itself repeats.
- Prefer a negated class to `.*` when you know the closing character.
- Use `fullmatch` for validation and `search` for finding.
- Escape any text that comes from outside with `re.escape`.
- Ask first whether a string method will do.

Challenges

EXERCISE 1

The pattern `^(\d+)+$` accepts strings made only of digits, but it is dangerous on a long string that ends in a non-digit. Write a safe pattern with the same meaning, and show that it returns quickly (under 0.1 seconds) on 5000 ones followed by `x`. Also check that the old and new patterns agree on the short inputs `""`, `"7"`, `"77"`, `"7a"`, `"a7"` and `"12 3"`. Do not run the old pattern on the long input.

View solution

EXERCISE 2

Given `<p>Hello <i>big</i> and <i>small</i> world, <i>end</i></p>`, first show what `<i>.*</i>` finds with `findall` and explain the problem. Then write a pattern using a negated class that returns just the three words inside the tags.

View solution

EXERCISE 3

Write `count_literal(needle, text)` that counts how many times a piece of text occurs, using `re.findall` and `re.escape`. Test it on the needles `1.5`, `(a+b)` and `a+b` against `Price 1.5, price 1x5, code (a+b) and (a+b) again, 1.5 end`, and print each count beside `text.count(needle)`. Then show how many matches the unescaped pattern `1.5` finds, and say which tool you would really use here.

View solution

QUICK REFERENCE

- **Catastrophic backtracking:** a repeat inside a repeat over the same characters, such as `(a+)+`; work roughly doubles per extra character on failure
- **Greedy overreach:** use `.*?` or, better, a negated class like `[^<]*`
- **match vs search:** `match` only tries the start of the string
- **$ and newline:** `$` allows a final newline; use `fullmatch` or `\Z`
- **\d:** matches other scripts' digits; use `[0-9]` or `re.A`
- **Alternation:** first working branch wins; put longer branches first
- **re.escape:** makes outside text literal inside a pattern
- **Not a pattern job:** fixed text (string methods) and nested structure (a parser)

COMING NEXT

Regular Expressions in Practice 10 is the capstone: a small set of reusable text helpers that combine what the whole course has covered.

Regular Expressions in Practice

Chapter 10 · Capstone: A Text Toolkit

The earlier chapters each taught one idea. In real programs the ideas arrive together, so this last chapter builds six small helpers that you could drop into any project. Each one is short, has a clear name, and uses patterns compiled once at module level (Chapter 6), with the habits from Chapter 9 built in.

> **WHAT WAS RUN**
>
> Every helper and every example below was executed, and the comments show exactly what Python printed.

Design rules for a toolkit

- **Compile once.** Patterns that never change live in module-level constants with an upper-case name.
- **One job per function.** A function takes text and returns a result. It does not print.
- **Name your groups.** `m["y"]` explains itself; `m[3]` does not.
- **Escape outside text.** Anything that did not come from you goes through `re.escape`.
- **Fail loudly when it matters.** A parser should raise an error on a bad line, not skip it silently.

Helper 1: squeeze whitespace

The simplest and most useful helper. It collapses every run of spaces, tabs and newlines into one space and trims the ends (Chapter 4).

```
import re

def squeeze(text):
    """Collapse runs of whitespace and trim the ends."""
    return re.sub(r"\s+", " ", text).strip()

print(repr(squeeze("  too   many \n spaces\there  ")))
```

```
# 'too many spaces here'
```

Helper 2: make a slug

A slug is the tidy, lower-case form of a title used in a web address. Instead of describing what to remove, describe what to *keep*: runs of letters and digits, then join them (Chapter 1).

```
def slugify(text):
    """Lower-case, keep letters and digits, join words with hyphens."""
    return "-".join(re.findall(r"[a-z0-9]+", text.lower()))

print(slugify("  Regular Expressions: in Practice! (2026) "))
# regular-expressions-in-practice-2026
```

Helper 3: find dates in two formats

This one combines alternation, named groups, `finditer` and a little ordinary Python (Chapters 2, 3, 6 and 7). A single pattern recognises both `2026-11-02` and `5 Dec 2026`. Because the two alternatives need different group names, the code checks which one matched. The pattern only finds the *shape*; the function then checks the month name against a dictionary, which is a job for code, not for a pattern.

```
MONTHS = {m: i for i, m in enumerate(
    ["jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec"]
DATE = re.compile(
    r"\b(?:(?P<y>\d{4})-(?P<m>\d{2})-(?P<d>\d{2})"
    r"|(?P<d2>\d{1,2})\s+(?P<mon>[A-Za-z]{3})[a-z]*\s+(?P<y2>\d{4}))\b")

def find_dates(text):
    """Return every date found, as YYYY-MM-DD, in order."""
    found = []
    for m in DATE.finditer(text):
        if m["y"]:
            y, mo, d = int(m["y"]), int(m["m"]), int(m["d"])
```

```
        else:
            mo = MONTHS.get(m["mon"].lower())
            if mo is None:
                continue
            y, d = int(m["y2"]), int(m["d2"])
        found.append(f"{y:04d}-{mo:02d}-{d:02d}")
    return found

print(find_dates("Sent 2026-11-02, due 5 Dec 2026 and 14 September 2027; bad: 3 Foo 2020
# ['2026-11-02', '2026-12-05', '2027-09-14']
```

The made-up month `3 Foo 2020` has the right shape, so the pattern matches it, but the dictionary lookup rejects it. Splitting the work this way keeps the pattern readable.

Helper 4: read key=value pairs

Values may be bare words or quoted text containing spaces. The alternation tries the quoted form first. The helper checks `is not None` rather than truthiness, so an empty quoted value is kept as an empty string instead of being lost.

```
PAIR = re.compile(r'(\w+)=(?:"([^"]*)"|(\S+))')

def parse_pairs(text):
    """Turn key=value pairs (values may be quoted) into a dict."""
    result = {}
    for m in PAIR.finditer(text):
        quoted, bare = m.group(2), m.group(3)
        result[m.group(1)] = quoted if quoted is not None else bare
    return result

print(parse_pairs('user=ann id=42 note="two words" empty=""'))
# {'user': 'ann', 'id': '42', 'note': 'two words', 'empty': ''}
```

Helper 5: censor whole words safely

The list of words comes from outside, so each is escaped. Longer words go first so that a short word cannot win inside a longer one (Chapter 9). The usual `\b` boundary breaks for a word that ends in a symbol, such as `c++`, because there is no word boundary between `+` and a space. Lookaround (Chapter 5) says what is wanted instead: not preceded and not followed by a letter, digit or underscore.

```
def censor(text, words):
    """Replace whole words, case-insensitively, with asterisks."""
    if not words:
        return text
    alt = "|".join(re.escape(w) for w in sorted(words, key=len, reverse=True))
    return re.sub(rf"(?<!\w)(?:{alt})(?!\w)", lambda m: "*" * len(m[0]), text, flags=re.

print(censor("The Cat sat; concatenate is safe, CAT again. C++ ok", ["cat", "c++"]))
# The *** sat; concatenate is safe, *** again. *** ok
```

Notice that `concatenate` is left alone, because `cat` inside it has letters on both sides. The empty-list check matters too: an empty alternation would build the pattern `(?:)`, which matches everywhere.

Helper 6: mask long numbers

A replacement function (Chapter 4) can compute its output from the match. Here every run of six or more digits keeps only its last four digits; shorter numbers are untouched.

```
def mask_digits(text, keep=4):
    """Hide all but the last `keep` digits of every run of 6+ digits."""
    return re.sub(r"\d{6,}", lambda m: "#" * (len(m[0]) - keep) + m[0][-keep:], text)

print(mask_digits("card 1234567890123456 ref 12345 id 9876543"))
# card ############3456 ref 12345 id ###6543
```

Using the helpers together

Because each helper takes text and returns text (or a simple value), they chain naturally:

```
note = "  Meeting   moved to 5 Dec 2026\n(was 2026-11-30).  Card 1234567890123456; ref=\
clean = squeeze(note)
print(clean)
print(find_dates(clean))
print(mask_digits(clean))
print(slugify(clean)[:30])
print(len(clean) < len(note))
# Meeting moved to 5 Dec 2026 (was 2026-11-30). Card 1234567890123456; ref="A 12"
# ['2026-12-05', '2026-11-30']
# Meeting moved to 5 Dec 2026 (was 2026-11-30). Card ############3456; ref="A 12"
# meeting-moved-to-5-dec-2026-wa
# True
```

HELPER	MAIN TOOLS FROM THE COURSE	CHAPTER
`squeeze`	`re.sub`, `\s+`	4
`slugify`	`findall`, character classes	1
`find_dates`	alternation, named groups, `finditer`	2, 3, 6, 7
`parse_pairs`	groups, alternation, `finditer`	2, 6
`censor`	`re.escape`, lookaround, replacement function, flags	3, 4, 5, 9
`mask_digits`	counted quantifier, replacement function	1, 4

Where to go next

- Add a test for every helper, including awkward inputs: empty text, a long near-miss, text full of special characters.
- Use `re.VERBOSE` and comments as soon as a pattern passes one line (Chapter 3).
- When the input has nested structure, reach for a real parser (Chapter 9).

- Read the standard library documentation for the `re` module. You now know enough to follow every section of it.

Challenges

EXERCISE 1

Write `redact_emails(text)` that replaces every email address with `[email]` and returns both the new text and the number of replacements. Test it on `Write to ann@example.com or bob.smith+x@mail.co.uk today.` and on `Nothing to hide here, only a@ and @b.`

View solution

EXERCISE 2

Write `top_words(text, n=3)` that returns the `n` most common words, ignoring case, with words such as `didn't` kept whole. Test it on `The cat and the hat. The CAT didn't sit; the hat didn't mind.` with the default and with `n=5`.

View solution

EXERCISE 3

Write `parse_config(text)` for a small settings format: `[section]` headers, `key = value` lines, blank lines and `#` comments (also allowed after a value). Return a dictionary of dictionaries. Use a `VERBOSE` pattern with named groups, and raise `ValueError` with the line number for an unreadable line or a key that appears before any section.

View solution

QUICK REFERENCE

- **Compile once:** module-level upper-case constants for fixed patterns
- **One job per function:** text in, result out, no printing
- **Named groups:** `m["name"]` over `m[2]`

- **Outside text:** always `re.escape`, and guard against an empty list
- **Boundaries for symbols:** `(?<!\w)` and `(?!\w)` when `\b` will not do
- **Pattern plus code:** let the pattern find the shape and ordinary code check the meaning
- **Parsers:** raise an error with the line number rather than skipping bad input

COURSE COMPLETE

That is the end of Regular Expressions in Practice. Keep the helpers from this chapter, add your own as you meet new problems, and always test with awkward input.