Module 5 of 20

Strings and text

Strings as immutable sequences, slicing and the method set, f-strings and the format-spec mini-language, code points versus bytes with the digit tests, the parsing shapes for every input format, and regular expressions with groups and substitution.

Text is what every program reads and writes, and Python's `str` — an immutable sequence of Unicode code points with the richest method set in the language — is why Python programs that handle text are short. This module makes that method set second nature, fixes the slice rule that lists and tuples share, and learns the format-spec mini-language once so that tables, fixed decimals and padded fields never cost a search again. It also draws the line between text and bytes that Python 3 enforces, and explains the three digit tests, case folding and normalisation that decide whether two strings are "the same".

String basics covers literals, escapes, raw strings, the sequence operations, comparison by code point and `repr`. Slicing and methods gives `[a:b:c]`, the case and whitespace methods, searching (`in`, `find`, `index`), splitting, joining, `partition`, `replace` and `translate`, the `is*` tests and padding. Formatting is f-strings and the spec grammar, floats, tables, `str.format` and the legacy `%`. Characters and Unicode covers `ord` / `chr`, `encode` / `decode`, `isdecimal` / `isdigit` / `isnumeric`, `casefold` and NFC. Parsing input gives a template for each input shape and the rule to convert strictly and reject clearly. Regular expressions covers the six `re` functions, the syntax, groups, substitution, greediness and flags.

The exercises are whole programs: a palindrome test over normalised text, text versus numeric ordering, a slicing command interpreter, a slug maker, an invoice table, a number-format row, a Caesar cipher through `translate`, a character report with byte lengths and digit tests, a typed configuration parser, a run-length codec written as a hand scanner, a number extractor and a log-line parser with named groups. The checkpoint adds a text-statistics table, a query-string parser with typed values and duplicate detection, and an access-log summary driven by one regular expression.

CONTENTS

1 String basics — an immutable sequence of characters 12 min

2	Slicing and the string methods	14 min
3	Formatting — f-strings and the format-spec mini-language	14 min
4	Characters, code points, bytes and Unicode	13 min
5	Parsing input — from lines and tokens to values	14 min
6	Regular expressions — the re module	15 min
✓	Checkpoint — Strings and text	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

String basics — an immutable sequence of characters

A Python `str` is an immutable sequence of Unicode characters. Each of those four words matters: *immutable* means every "change" makes a new string; *sequence* means indexing, slicing, `len`, `in` and iteration all work as they do on a list; *Unicode* means a character is a code point, not a byte, and `"é"` has length 1; and *string* means the type comes with the richest method set in the language. This lesson covers literals and escapes, the sequence operations, comparison, `str` versus `repr`, and the concatenation cost that decides how output should be built.

Literals

```
a = 'single'
b = "double"                      # same thing; pick one and be consistent
c = "it's"                        # the other quote avoids escaping
d = 'say "hi"'
e = "line one\nline two\ttabbed"  # escapes: \n newline, \t tab, \\ backslash,
f = r"C:\new\folder"              # raw: backslashes are literal (regexes, Wi
g = """A multi-line
string keeps its newlines."""
h = "adjacent " "literals " "join"   # implicit concatenation at compile time
```

Escapes are interpreted in ordinary literals: `\n`, `\t`, `\\`, `\'`, `\"`, `\x41` (hex byte), `\u00e9` (Unicode code point), `\N{EURO SIGN}`. A raw string turns them off — except that a raw string still cannot end in an odd backslash. Triple-quoted strings span lines and are the docstring syntax. Adjacent literals are joined by the compiler, which is how a long message is split across lines inside parentheses without `+`.

Sequence operations

```
s = "python"
len(s)          # 6
s[0], s[-1]     # 'p', 'n'  — negative indexes count from the end
s[2:4]          # 'th'      — slicing, half-open (next lesson)
"th" in s       # True      — substring test
s + "3"         # 'python3'
"-" * 10        # '----------'
for ch in s:    # iterates characters
    ...
list(s)         # ['p', 'y', 't', 'h', 'o', 'n']
```

Indexing past the end raises `IndexError`; slicing past the end does not (it clips). There is no separate character type — `s[0]` is a string of length 1. `in` tests for a substring, not just a character: `"tho" in "python"` is `True`.

Immutability

```
s = "cat"
s[0] = "b"          # TypeError: 'str' object does not support item assignment
s = "b" + s[1:]     # 'bat' — a new string bound to the same name
```

Every method that appears to modify — `upper`, `replace`, `strip` — returns a new string and leaves the original alone; the classic bug is calling one and discarding the result (`s.upper()` on its own line does nothing). Immutability is what makes strings usable as dictionary keys and set members, and it is why building a long string by repeated `+=` is expensive: each step copies the whole accumulated string. Collect the pieces in a list and `"".join` them once (Module 3 lesson 5).

Comparison

Strings compare by code point, left to right, first difference decides; a prefix sorts before the longer string:

```
"apple" < "banana"    # True
"Zebra" < "apple"     # True — uppercase letters have smaller code points thar
"abc" < "abd"         # True
"ab" < "abc"          # True
"10" < "9"            # True — text, not numbers
```

Case-insensitive comparison is `a.lower() == b.lower()` — or, correctly for every language, `a.casefold() == b.casefold()`. Comparing numeric strings as numbers means converting them first: `sorted(nums, key=int)`. `==` compares content, never identity; two equal strings may or may not be the same object, and `is` is not for strings.

---

str and repr

`str(x)` is the readable form, what `print` shows; `repr(x)` is the unambiguous form, what the REPL shows and what you want in a debug message:

```
s = "a\tb\n"
print(str(s))     # a   b   (a tab and a newline, invisible)
print(repr(s))    # 'a\tb\n'
print(f"{s!r}")   # the same, inside an f-string
```

`repr` of a string is a valid literal that would recreate it, quotes and escapes included. When a program's output is wrong by "nothing visible", `repr` shows the trailing newline or the tab.

The methods, first pass

The next lesson goes through them; the ones every program needs are `strip()` (remove surrounding whitespace), `split()` (to a list of words), `join(iterable)` (from a list back to a string), `upper()` / `lower()`, `replace(old, new)`, `startswith` / `endswith`, `find`, and `count`. Note the direction of `join`: it is a method of the *separator* — `", ".join(names)` — because the separator is the one string that is always a string, while the pieces might be any iterable.

Pitfalls

- `s.strip()` (or any method) called without keeping the result.
- `s[0] = "x"`. Build a new string.

`"10" < "9"` and `sorted(["10", "9"])` — text order, not numeric.

`+=` in a loop over many pieces; use a list and `join`.

Treating `s[i]` as a char type: it is a `str`, and `s[i] == "a"` is the comparison.

Confusing `str` and `repr` when printing a debug value.

Key takeaways

Strings are immutable sequences of code points: index, slice, `len`, `in`, iterate; every change is a new string.

Literals: single or double quotes, escapes, raw strings for backslashes, triple quotes for multi-line, adjacent literals join.

Comparison is by code point (uppercase before lowercase, text order for digits); `casefold` for case-insensitive.

`str` is readable, `repr` is unambiguous — use `repr` when debugging whitespace.

`sep.join(pieces)` builds a string; repeated `+=` copies every time.

EXERCISE 1

Palindromes, normalised

Read lines until the end of input and decide whether each is a palindrome when only letters and digits are considered and case is ignored: build the normalised string with `casefold()` and a filter on `isalnum()`, then compare it with its reverse `[::-1]`.

Input: lines of text. **Output:** `yes` or `no` per line.

```
A man, a plan, a canal: Panama
hello
```

prints

```
yes
no
```

INPUT 1

```
A man, a plan, a canal: Panama
hello
```

OUTPUT 1

```
yes
no
```

REFERENCE SOLUTION

```
import sys


def is_palindrome(text):
    core = "".join(ch for ch in text.casefold() if ch.isalnum())
    return core == core[::-1]


for line in sys.stdin:
    print("yes" if is_palindrome(line.rstrip("\n")) else "no")
```

EXERCISE 2

Text order versus numeric order

Strings compare by code point, which is not numeric order and puts uppercase before lowercase. Read a line of tokens and print them sorted as text; then, if every token parses as an integer, print them sorted numerically as well (using `key=int`), otherwise print `numeric: n/a`.

Input: one line of tokens. **Output:** `text: <tokens>` then `numeric: <tokens>` or `numeric: n/a`.

```
10 9 100
```

prints

```
text: 10 100 9
numeric: 9 10 100
```

INPUT 1

```
10 9 100
```

OUTPUT 1

```
text: 10 100 9
numeric: 9 10 100
```

REFERENCE SOLUTION

```
tokens = input().split()
print("text:", " ".join(sorted(tokens)))
try:
    numeric = sorted(tokens, key=int)
except ValueError:
    print("numeric: n/a")
else:
    print("numeric:", " ".join(numeric))
```

QUIZ

1. What does this do?

```
s = "cat"
s[0] = "b"
```

A. `s` becomes `"bat"`

B. `TypeError` — strings are immutable

C. `s` becomes `"bcat"`

D. `IndexError`

2. What is `"Zebra" < "apple"`?

A. `False`

B. `True`

C. `TypeError`

D. Depends on the locale

3. What does `print(repr("a\tb"))` print?

A. `a b`

B. `'a\tb'`

C. `a\tb`

D. `"a\tb"`

4. Why is `"".join(parts)` preferred to `s += part` in a loop?

A. Because `+=` is not defined for strings

B. Because each `+=` copies the whole string so far, making the loop quadratic

C. Because `join` deduplicates

D. There is no difference

5. What is `r"\n"`?

A. A newline character

B. A two-character string: backslash and `n`

C. A syntax error

D. An empty string

ANSWER KEY · STRING BASICS — AN IMMUTABLE SEQUENCE OF CHARACTERS

1. B A string cannot be changed in place; build a new one with `"b" + s[1:]`. Every string method likewise returns a new string.

2. B Comparison is by code point, and uppercase letters (65–90) come before lowercase (97–122). Use `key=str.casefold` for a case-insensitive order.

3. B `repr` returns a quoted literal with escapes visible, which is why it is the form to print when debugging whitespace. `str` would show the actual tab.

4. B Immutability means concatenation allocates a new string every time; `join` measures the total once and copies each piece once.

5. B A raw literal leaves backslashes alone, which is what regular expressions and Windows paths need. Its length is 2.

LESSON 2 · 14 MIN

Slicing and the string methods

Two things make Python string handling short: slicing, which extracts any substring, reversal or every-other-character in one expression, and a method set that covers case, whitespace, searching, splitting, joining, testing and padding without a library. This lesson gives the slice rule precisely — it is the same rule for lists and tuples — and then the methods grouped by what they do, with the return-value conventions (`find` returns -1, `index` raises) that decide which one to call.

Slicing

`s[start:stop:step]` — `start` included, `stop` excluded, `step` the stride; each part optional; negative indexes count from the end; out-of-range bounds clip rather than raise.

```
s = "abcdefg"
s[1:4]      # 'bcd'
s[:3]       # 'abc'    — from the start
s[3:]       # 'defg'   — to the end
s[-3:]      # 'efg'    — last three
s[:-1]      # 'abcdef' — all but the last
s[::2]      # 'aceg'   — every second
s[::-1]     # 'gfedcba' — reversed
s[1:100]    # 'bcdefg' — clipped, no error
s[5:2]      # ''       — empty, no error
```

The half-open rule means `s[:k] + s[k:] == s` for any `k`, and the length of `s[a:b]` is `b - a` when in range. `s[::-1]` is the idiom for reversing a string (and `"".join(reversed(s))` the readable alternative). A slice always returns a new string; slicing a list returns a new list (Module 6) — the syntax is identical.

Case and whitespace

```
"Hello World".lower()        # 'hello world'
"hello".upper()              # 'HELLO'
"hello world".title()        # 'Hello World'
"hello".capitalize()         # 'Hello'
"Straße".casefold()          # 'strasse' — the aggressive lowercase for compar
"  pad  ".strip()            # 'pad'
"  pad  ".lstrip()           # 'pad  '
"xxhixx".strip("x")          # 'hi'  — the argument is a *set* of characters,
"file.txt".removesuffix(".txt")   # 'file' (3.9); removeprefix likewise
```

`strip()` with no argument removes all whitespace (spaces, tabs, newlines) from both ends. With an argument it removes any of those characters, repeatedly, from both ends — `"www.example.com".strip("w.")` gives `'example.com'`, and `"hello.py".strip(".py")` gives `'hello'` only by luck (`'happy.py'.strip('.py')` gives `'ha'`). For a literal prefix or suffix, `removeprefix` / `removesuffix`.

Searching

```
s = "banana"
s.find("an")        # 1   — first index, or -1
s.rfind("an")       # 3   — last
s.index("an")       # 1   — like find, but raises ValueError when absent
s.count("an")       # 2   — non-overlapping
s.startswith("ba")  # True
s.endswith(("na", "xa"))    # True — a tuple tests several suffixes
"an" in s           # True — the existence test; cheaper than find ≠ -1
```

Use `in` when you only need yes/no; `find` when you need the position and absence is normal; `index` when absence is a bug you want raised. `find` and friends take optional `start` / `end` bounds: `s.find("a", 2)`.

Splitting and joining

```
"a b  c".split()             # ['a', 'b', 'c']   — any whitespace, runs colla
"a,b,,c".split(",")          # ['a', 'b', '', 'c'] — exact separator, empties
"a,b,c".split(",", 1)        # ['a', 'b,c']       — at most 1 split
"a,b,c".rsplit(",", 1)       # ['a,b', 'c']       — from the right
"k=v=w".partition("=")       # ('k', '=', 'v=w')  — always a 3-tuple; ('k=v=w
"one\ntwo\n".splitlines()    # ['one', 'two']     — no trailing empty element
", ".join(["a", "b"])        # 'a, b'
"".join(reversed("abc"))     # 'cba'
```

`split()` with no argument is the tokeniser for whitespace-separated input; `split(sep)` is for delimited fields where empty fields matter (CSV-like lines). `partition` is the tool for "the part before the first `=` and the part after" and never raises. `join` requires an iterable of *strings*: `",".join([1, 2])` is a `TypeError`, so map with `str` first.

Replacing and translating

```
"a-b-c".replace("-", "+")        # 'a+b+c'
"a-b-c".replace("-", "+", 1)     # 'a+b-c' — at most once
table = str.maketrans("abc", "xyz")
"aabbcc".translate(table)        # 'xxyyzz' — character-for-character, in one
"hello".translate(str.maketrans("", "", "lo"))   # 'he' — the third argument
```

`replace` is for substrings; `translate` is faster for many single-character substitutions or deletions (removing punctuation, for instance) because it makes one pass with a table.

Testing

```
"123".isdigit()      # True
"abc".isalpha()      # True
"ab1".isalnum()      # True
"  ".isspace()       # True
"Hello".isupper()    # False
"hello".islower()    # True
"Hello World".istitle()   # True
```

`isdigit` is true for `"²"` and false for `"-1"` and `"1.5"`, so it is a check for "all digit characters", not for "parses as a number" — for that, try `int()` and catch `ValueError`

(Module 2 lesson 5). The next lesson separates `isdigit`, `isdecimal` and `isnumeric`.

Padding and alignment

```
"42".zfill(5)          # '00042'
"-42".zfill(5)         # '-0042' — sign-aware
"ab".ljust(5, ".")     # 'ab...'
"ab".rjust(5)          # '   ab'
"ab".center(6, "*")    # '**ab**'
```

Format specs (next lesson) do the same inside an f-string — `f"{'ab':.<5}"` — and are what you use when the padding is part of a larger line.

Pitfalls

`strip(".txt")` as a suffix remover. It strips *characters*; use `removesuffix`.

`split(" ")` for whitespace tokenising — it keeps empty strings between consecutive spaces; use `split()`.

`index` on a value that may be absent, raising `ValueError`; use `find` or `in`.

`join` on a list of non-strings.

`s.find(x) == True` — `find` returns a position; `1 == True` but `0 ≠ True`.

Expecting `title()` to handle apostrophes (`"it's"` → `"It'S"`).

Key takeaways

`s[a:b:c]` — half-open, negative indexes from the end, clipping bounds, `[::-1]` reverses.

`strip` / `lstrip` / `rstrip` remove characters from a set at the ends; `removeprefix` / `removesuffix` remove a literal.

`in` for existence, `find` for a position or −1, `index` to raise; `count`, `startswith` / `endswith` with tuples.

`split()` tokenises whitespace, `split(sep)` keeps empties, `partition` is a 3-tuple, `join` needs strings.

`replace` for substrings, `translate` for many characters; `is*` test character classes; `zfill` / `ljust` / `rjust` / `center` pad.

EXERCISE 1

Slice commands

Read a string, then `n` commands, and print the result of each slice: `rev` reverses; `every k` takes every k-th character from the start; `first k` and `last k` take that many characters; `drop k` removes the first k; `mid` removes the first and last character. Every answer is one slice expression.

Input: the string, then `n`, then `n` commands. **Output:** one line per command (an empty result prints `(empty)`).

```
abcdefg
4
rev
every 2
last 3
drop 2
```

prints

```
gfedcba
aceg
efg
cdefg
```

INPUT 1

```
abcdefg
4
rev
every 2
last 3
drop 2
```

OUTPUT 1

```
gfedcba
aceg
efg
cdefg
```

REFERENCE SOLUTION

```python
s = input()
n = int(input())
for _ in range(n):
    cmd, *arg = input().split()
    k = int(arg[0]) if arg else 0
    if cmd == "rev":
        result = s[::-1]
    elif cmd == "every":
        result = s[::k]
    elif cmd == "first":
        result = s[:k]
    elif cmd == "last":
        result = s[-k:] if k else ""
    elif cmd == "drop":
        result = s[k:]
    else:
        result = s[1:-1]
    print(result if result else "(empty)")
```

EXERCISE 2

Slugs

Turn titles into URL slugs without regular expressions: casefold the text, replace every character that is not a letter or digit with a hyphen, collapse runs of hyphens into one, and strip hyphens from both ends. Build the result character by character or with `split` / `join`.

Input: lines of titles. **Output:** one slug per line, or `(empty)` when nothing remains.

```
Hello, World!
  Python  3.11 is here
```

prints

```
hello-world
python-3-11-is-here
```

INPUT 1

```
Hello, World!
  Python  3.11 is here
```

OUTPUT 1

```
hello-world
python-3-11-is-here
```

REFERENCE SOLUTION

```python
import sys


def slugify(title):
    replaced = "".join(ch if ch.isalnum() else "-" for ch in title.casefold())
    return "-".join(part for part in replaced.split("-") if part)


for line in sys.stdin:
    slug = slugify(line.rstrip("\n"))
    print(slug if slug else "(empty)")
```

QUIZ

1. What is `"abcdef"[1:5:2]`?

A. `'bd'`

B. `'bdf'`

C. `'ace'`

D. `'bcde'`

2. What does `"happy.py".strip(".py")` return?

A. `'happy'`

B. `'ha'`

C. `'happy.py'`

D. `ValueError`

3. What is `"a b c".split(" ")`?

A. `['a', 'b', 'c']`

B. `['a', 'b', '', 'c']`

C. `['a', 'b', ' ', 'c']`

D. `['a b c']`

4. Which expression tells you *where* `"x"` occurs in `s`, giving −1 when absent?

A. `s.index("x")`

B. `s.find("x")`

C. `"x" in s`

D. `s.count("x")`

5. What does `"-".join([1, 2, 3])` do?

A. Returns `'1-2-3'`

B. Raises `TypeError` — join needs strings

C. Returns `'[1, 2, 3]'`

D. Returns `'-1-2-3-'`

6. What is `"k=v=w".partition("=")`?

A. `('k', 'v', 'w')`

B. `('k', '=', 'v=w')`

C. `['k', 'v=w']`

D. `('k=v', '=', 'w')`

ANSWER KEY · SLICING AND THE STRING METHODS

1. A Start at index 1 (`b`), step 2, stop before index 5: indexes 1 and 3 — `b` and `d` .

2. B `strip` removes any of the given *characters* from both ends repeatedly: `p` , `y` and `.` are stripped until `a` stops it. `removesuffix(".py")` removes the literal suffix.

3. B With an explicit separator, consecutive separators produce empty strings. The no-argument `split()` collapses whitespace runs.

4. B `find` returns the position or −1; `index` raises `ValueError` when absent; `in` gives a boolean; `count` gives how many.

5. B `join` accepts only an iterable of strings. Convert first: `"-".join(map(str, [1, 2, 3]))` .

6. B `partition` splits at the first occurrence and always returns a 3-tuple including the separator; `rpartition` splits at the last.

LESSON 3 · 14 MIN

Formatting — f-strings and the format-spec mini-language

Every judged exercise ends with output that must match exactly, and almost every real program prints a table, a report or a message with values in it. Python's f-string does both: an expression inside braces, an optional conversion, and a *format spec* after a colon that controls width, alignment, padding, sign, grouping, precision and type. The spec is a small language shared by `format()`, `str.format` and f-strings, and this lesson learns it once. It ends with the older `%` style, which you will read in existing code and should not write.

f-strings

```
name, n, price = "Ada", 3, 2.5
print(f"{name} bought {n} items for {n * price}")   # any expression inside th
print(f"{name!r} {name!s} {name!a}")                # conversions: repr, str,
print(f"{n=}, {price=}")                            # 3.8: name=value, for deb
print(f"{'nested'}")                                # a different quote style
print(f"{{literal braces}}")                        # doubled braces print one
```

The braces take an expression, so calls, indexing, arithmetic and conditional expressions all work — but keep them short; a complex expression belongs in a variable above the f-string. `!r` applies `repr`, which is how a string is printed with its quotes. Until 3.12 the same quote character as the enclosing string cannot appear inside the braces; use the other one.

The format spec

`{value:spec}` where `spec` is `[[fill]align][sign][#][0][width][grouping][.precision][type]`, every part optional:

Part	Values	Effect
fill + align	any char, then `<` `>` `^` `=`	pad to the width: left, right, centre; `=` puts padding after the sign
sign	`+` `-` ` `	`+` shows a plus on positives; space leaves a space
`#`		alternate form: `0x` prefix, `0b`, a decimal point on `.0f`
`0`		zero-pad (numbers) — shorthand for `fill=0, align=`
width	integer	minimum field width
grouping	`,` `_`	thousands separator
`.precision`	integer	decimals for `f`, significant digits for `g`, max chars for strings
type	`d` `f` `e` `g` `%` `x` `X` `o` `b` `s` `c`	how to render

```
f"{42:5d}"        # '   42'      width 5, right-aligned (numbers default right
f"{42:<5d}|"      # '42   |'     left
f"{42:^7d}|"      # '  42   |'   centred
f"{42:05d}"       # '00042'      zero-padded
f"{-42:05d}"      # '-0042'
f"{42:+d}"        # '+42'
f"{1234567:,}"    # '1,234,567'
f"{1234567:_d}"   # '1_234_567'
f"{255:x} {255:X} {255:#x} {5:08b}"   # 'ff FF 0xff 00000101'
f"{'ab':>5}|"     # '   ab|'     strings default left; > forces right
f"{'ab':*^6}"     # '**ab**'     fill character then align
f"{'abcdef':.3}"  # 'abc'        precision truncates a string
```

The width is a *minimum*: a value wider than the field is printed whole. The width and precision can themselves be expressions: `f"{x:{w}.{p}f}"`.

Floats

```
x = 3.14159
f"{x:.2f}"        # '3.14'       fixed, 2 decimals — the one every exercise us
f"{x:8.3f}"       # '   3.142'   width 8
f"{x:e}"          # '3.141590e+00'
f"{x:.3g}"        # '3.14'       3 significant digits; switches to e-notation
f"{0.256:.1%}"    # '25.6%'      multiplies by 100
f"{2.0:.0f}"      # '2'
f"{1e6:,.2f}"     # '1,000,000.00'
f"{x}"            # '3.14159'    no spec: repr-like shortest form
```

`.Nf` rounds correctly to the nearest representable value (half-to-even on exact ties, but exact ties are rare in binary — Module 2). A float with no spec prints the shortest round-tripping form, which is never what a table wants; always give a precision.

Tables

```
rows = [("apple", 3, 0.5), ("watermelon", 1, 4.25)]
print(f"{'item':<12}{'qty':>4}{'price':>8}")
for name, qty, price in rows:
    print(f"{name:<12}{qty:>4}{price:>8.2f}")
```

```
item         qty   price
apple          3    0.50
watermelon     1    4.25
```

Column widths are decided once; the same specs apply to the header (strings) and the rows. When the widest value is only known after reading the data, compute `w = max(len(r[0]) for r in rows)` and use `{name:<{w}}`.

---

format() and str.format

`format(value, spec)` applies one spec to one value: `format(255, "08b")`. `"{} and {}".format(a, b)` is the pre-3.6 template style, still useful when the template is built separately from the values (a message loaded from a file, a format reused in a loop):

```
template = "{name:<10}{score:>5}"
for name, score in rows:
    print(template.format(name=name, score=score))
```

Positional `{0}`, named `{name}`, and attribute/index access `{p.x}`, `{d[key]}` all work inside `str.format`.

The % operator

`"%s scored %d (%.2f%%)" % (name, score, pct)` is the C-style formatting Python inherited; `%s` for anything, `%d` for integers, `%f` for floats, `%%` for a literal percent. It is what `logging` messages use (Module 15), because the formatting is deferred until the message is actually emitted, and it is everywhere in older code. Read it; write f-strings.

Pitfalls

A float printed without a precision.

`f"{x:5}"` for a string aligns *left*, for a number *right*; be explicit with `<` or `>`.

Expecting width to truncate: it does not; only `.precision` truncates strings.

The same quote inside the braces on 3.11 (`f"{d["k"]}"` is a `SyntaxError` there; use `d['k']`).

`f"{x:,.2f}"` versus `f"{x:.2f,}"` — grouping comes before precision.

`%` formatting with a single tuple argument: `"%s" % (1, 2)` is a `TypeError`; wrap it `((1, 2),)`.

Key takeaways

f-strings evaluate expressions in braces; `!r` for repr, `=` for name=value, `{{ }}` for literal braces.

The spec is `[fill][align][sign][#][0][width][,][.precision][type]`; numbers align right by default, strings left.

`.2f` is the fixed-decimal spec; `,` groups thousands; `%` scales by 100; `x` / `b` / `o` change the base.

Tables are the same specs applied to header and rows; widths can be variables.

`str.format` is for templates separated from values; `%` is legacy — read it, do not write it.

EXERCISE 1

Invoice table

Read `n` rows `item qty price` and print an aligned table: a header, one row per item, and a total line. Columns: item left-aligned in 10, qty right-aligned in 5, price and line total right-aligned in 10 with two decimals. The total line puts `TOTAL` in the item column and the grand total in the last column, with the middle columns blank.

Input: `n`, then `n` lines. **Output:** `n + 2` lines.

```
2
apple 3 0.5
melon 1 4.25
```

prints

```
item        qty     price     total
apple         3      0.50      1.50
melon         1      4.25      4.25
TOTAL                          5.75
```

INPUT 1

```
2
apple 3 0.5
melon 1 4.25
```

OUTPUT 1

```
item        qty     price     total
apple         3      0.50      1.50
melon         1      4.25      4.25
TOTAL                          5.75
```

REFERENCE SOLUTION

```python
n = int(input())
print(f"{'item':<10}{'qty':>5}{'price':>10}{'total':>10}")
grand = 0.0
for _ in range(n):
    item, qty, price = input().split()
    qty, price = int(qty), float(price)
    line = qty * price
    grand += line
    print(f"{item:<10}{qty:>5}{price:>10.2f}{line:>10.2f}")
print(f"{'TOTAL':<10}{'':>5}{'':>10}{grand:>10.2f}")
```

EXERCISE 2

Number formats

For each integer print five renderings on one line separated by single spaces: decimal, lowercase hexadecimal, binary zero-padded to width 8, decimal zero-padded to width 5, and decimal with thousands separators.

Input: `n`, then `n` integers. **Output:** one line per integer.

```
2
255
1234567
```

prints

```
255 ff 11111111 00255 255
1234567 12d687 10010110101101000 0111 1234567 1,234,567
```

INPUT 1

```
2
255
1234567
```

OUTPUT 1

```
255 ff 11111111 00255 255
1234567 12d687 10010110101101000 0111
1234567 1,234,567
```

REFERENCE SOLUTION

```python
n = int(input())
for _ in range(n):
    x = int(input())
    print(f"{x:d} {x:x} {x:08b} {x:05d} {x:,}")
```

QUIZ

1. What does `f"{42:06d}"` produce?

A. `'000042'`

B. `'42 '`

C. `' 42'`

D. `'42'`

2. What does `f"{'ab':5}|"` produce?

A. `' ab|'`

B. `'ab |'`

C. `'ab|'`

D. `' ab |'`

3. What is `f"{0.256:.1%}"`?

A. `'0.3%'`

B. `'25.6%'`

C. `'26%'`

D. `'0.256%'`

4. What does `f"{x=}"` do for `x = 5`?

A. Raises `SyntaxError`

B. Produces `'x=5'`

C. Assigns 5 to the string

D. Produces `'5'`

5. Which spec prints `1234567.891` as `1,234,567.89`?

A. `{:.2f,}`

B. `{:,.2f}`

C. `{:2,f}`

D. `{:,2f}`

6. How is a literal `{` written inside an f-string?

A. `\{`

B. `{{`

C. `{'{'}`

D. It cannot be

ANSWER KEY · FORMATTING — F-STRINGS AND THE FORMAT-SPEC MINI-LANGUAGE

1. A The `0` flag zero-pads a number to the width. Width alone would right-align with spaces.

2. B Strings align left by default (numbers right). Use `>` to right-align a string.

3. B The `%` type multiplies by 100, appends a percent sign and applies the precision to the decimals.

4. B The `=` specifier (3.8) prints the expression text and its value — a debugging convenience.

5. B Grouping comes before precision in the spec grammar: `[width][,][.precision][type]`.

6. B Doubled braces print one brace. (`{'{'}` also works, as an expression, but the doubled form is the convention.)

LESSON 4 · 13 MIN

Characters, code points, bytes and Unicode

A Python string holds *code points* — abstract Unicode characters numbered from 0 to 1 114 111 — and a `bytes` object holds bytes. The two are different types, converted by *encoding* (str → bytes) and *decoding* (bytes → str) with a named encoding, almost always UTF-8. Keeping the distinction straight is the whole of Unicode handling in Python 3, and it dissolves the classic questions: why `len("é")` is 1 but the file is two bytes longer, why `"ß".upper()` is `"SS"`, why `"²".isdigit()` is true. This lesson covers `ord` / `chr`, the text/bytes boundary, the three digit tests, case and normalisation, and the `string` module constants.

Code points

```
ord("A")        # 65
ord("é")        # 233
ord("€")        # 8364
chr(65)         # 'A'
chr(0x1F600)    # '😀'
"\u00e9"        # 'é'  — escape by code point
"\N{GREEK SMALL LETTER ALPHA}"   # 'α' — by name
```

`ord` and `chr` convert between a one-character string and its code point; arithmetic on code points is how you shift letters — `chr(ord("a") + 3)` is `"d"` — and the Caesar cipher is the standard exercise. `unicodedata.name("é")` gives `'LATIN SMALL LETTER E WITH ACUTE'`, `unicodedata.category("é")` gives `'Ll'` (letter, lowercase). ASCII is the first 128 code points; everything you type on an English keyboard is there, and `"é"`, `"€"`, CJK characters and emoji are above it.

str and bytes

```
s = "café"
b = s.encode("utf-8")       # b'caf\xc3\xa9'
len(s), len(b)              # 4, 5 — é is two bytes in UTF-8
b.decode("utf-8")           # 'café'
b.decode("ascii")           # UnicodeDecodeError: byte 0xc3 is not ASCII
"é".encode("latin-1")       # b'\xe9' — a different encoding, one byte
```

A `bytes` literal is `b"..."` and may contain only ASCII; other values are written as `\x..` escapes. `bytes` supports most string methods (`split`, `strip`, `find`, `upper` on ASCII) and indexes to integers: `b"abc"[0]` is `97`. Files opened in text mode do the encoding for you (`open(path, encoding="utf-8")`, Module 14); sockets and binary files hand you bytes. The rule: **decode at the boundary, work in `str`, encode at the boundary.** Mixing the types is a `TypeError` (`"a" + b"b"`), which is a feature: Python 2's silent mixing was the source of every mojibake bug.

UTF-8 encodes ASCII in one byte, most European letters in two, CJK in three, emoji in four, and is the default nearly everywhere — including `sys.stdin` / `sys.stdout` on the judge. `sys.getdefaultencoding()` is `'utf-8'`; the *file system* and *console* encodings can differ on Windows, which is why `open` should always be given `encoding=`.

Three digit tests

Method	`"5"`	`"²"`	`"½"`	`"٣"` (Arabic-Indic 3)	`"-1"`
`isdecimal()`	True	False	False	True	False
`isdigit()`	True	True	False	True	False
`isnumeric()`	True	True	True	True	False

`isdecimal` is "characters `int()` accepts as digits" — and `int("٣")` really is 3. `isdigit` adds superscripts and other digit-like symbols; `isnumeric` adds fractions and CJK numerals. None accepts a sign or a decimal point, so none is a test for "is a number"; that test is `try: int(s)` (Module 2). `isalpha` is true for letters in any script; `isascii()` (3.7) is true when every character is below 128.

Case is not simple

```
"straße".upper()          # 'STRASSE' — one character becomes two
"İ".lower()               # 'i' — two code points
"ǆ".title()               # 'ǆ' — a titlecase letter that is neither upper nor
"MASSE".lower() == "masse"          # True
"Maße".lower() == "masse"           # False
"Maße".casefold() == "masse"        # True — casefold is for comparison
```

`lower()` and `upper()` are for display; `casefold()` is for comparing and for dictionary keys that should ignore case. A string's length can change under a case mapping, so never index into `s.upper()` with positions from `s`.

Normalisation

The same visible character can be one code point (`"é"`, U+00E9) or two (`"e"` + U+0301, a combining acute). They print identically and compare unequal:

```
import unicodedata
a = "\u00e9"
b = "e\u0301"
a == b                                          # False
unicodedata.normalize("NFC", b) == a            # True — compose
len(unicodedata.normalize("NFD", a))            # 2 — decompose
```

Normalise (NFC) text that comes from users or files before comparing, deduplicating or hashing it. `unicodedata.normalize("NFKD", s)` followed by `s.encode("ascii", "ignore")` is the quick way to strip accents for a slug.

The string module and character arithmetic

`string.ascii_lowercase`, `ascii_uppercase`, `ascii_letters`, `digits`, `punctuation`, `whitespace` are the ready-made alphabets. Rotating within one:

```
import string
def caesar(text, k):
    lower = string.ascii_lowercase
    table = str.maketrans(lower + lower.upper(), lower[k:] + lower[:k] + (lowe
    return text.translate(table)
```

`translate` with a table from `maketrans` handles every character in one pass and leaves characters not in the table (spaces, digits, punctuation) unchanged.

Pitfalls

- `len(s)` as the byte length. It is the code-point count; `len(s.encode())` is bytes.

`open(path)` without `encoding="utf-8"` — the platform default may not be UTF-8.

`isdigit()` as "is a number".

`lower()` for comparison; `casefold()`.

Comparing user input without NFC normalisation.

`"a" + b"b"`. Decode first.

Key takeaways

A `str` is code points, a `bytes` is bytes; `encode` / `decode` with UTF-8 convert at the boundary and never mix.

`ord` / `chr` map characters to code points; `unicodedata` names and normalises them.

`isdecimal` ⊂ `isdigit` ⊂ `isnumeric`; none accepts a sign — parse with `int()` and catch `ValueError`.

`lower` / `upper` can change length; `casefold` is the comparison form.

Normalise (NFC) text from outside before comparing; `string` holds the alphabets; `translate` maps many characters in one pass.

EXERCISE 1

Caesar cipher

Shift every ASCII letter by `k` positions (wrapping around the alphabet), preserving case and leaving every other character alone. Build a translation table once with `str.maketrans` and apply it with `translate`. `k` may be negative or larger than 26.

Input: `k`, then lines of text. **Output:** each line shifted.

```
3
Hello, World!
```

prints

```
Khoor, Zruog!
```

INPUT 1

```
3
Hello, World!
```

OUTPUT 1

```
Khoor, Zruog!
```

REFERENCE SOLUTION

```
import string
import sys

k = int(input()) % 26
lower = string.ascii_lowercase
upper = string.ascii_uppercase
shifted_lower = lower[k:] + lower[:k]
shifted_upper = upper[k:] + upper[:k]
table = str.maketrans(lower + upper, shifted_lower + shifted_upper)
for line in sys.stdin:
    print(line.rstrip("\n").translate(table))
```

EXERCISE 2

Character report

For each whitespace-separated token print its length in characters, its length in bytes when encoded as UTF-8, and the results of `isdecimal()` and `isdigit()`.

Input: one line of tokens. **Output:** `<token> chars=<n> bytes=<n> decimal=<bool> digit=<bool>` per token.

```
café 42
```

prints

```
café chars=4 bytes=5 decimal=False digit=False
42 chars=2 bytes=2 decimal=True digit=True
```

INPUT 1

```
café 42
```

OUTPUT 1

```
café chars=4 bytes=5 decimal=False
digit=False
42 chars=2 bytes=2 decimal=True
digit=True
```

REFERENCE SOLUTION

```
for tok in input().split():
    print(f"{tok} chars={len(tok)} bytes={len(tok.encode('utf-8'))} decimal={tok.isc
```

QUIZ

1. What are `len("é")` and `len("é".encode("utf-8"))`?

A. `1` and `1`

B. `1` and `2`

C. `2` and `2`

D. `2` and `1`

2. What does `"a" + b"b"` do?

A. Returns `"ab"`

B. Returns `b"ab"`

C. Raises `TypeError`

D. Returns `"ab"` after decoding

3. Which method should be used to compare strings case-insensitively across languages?

A. `lower()`

B. `casefold()`

C. `upper()`

D. `title()`

4. What is `"²".isdigit()` and `"²".isdecimal()`?

A. `True`, `True`

B. `True`, `False`

C. `False`, `False`

D. `False`, `True`

5. Why can `"\u00e9" == "e\u0301"` be `False` even though both display as é?

A. Because one is a byte string

B. Because they are different code-point sequences — a precomposed character versus a base plus a combining accent; normalise with NFC

C. Because string comparison is by length only

D. It is always `True`

ANSWER KEY · CHARACTERS, CODE POINTS, BYTES AND UNICODE

1. B A `str` counts code points — one — while UTF-8 encodes U+00E9 in two bytes.

2. C Text and bytes never mix implicitly in Python 3; decode the bytes first. The strictness prevents the encoding bugs Python 2 was known for.

3. B `casefold` applies the full Unicode case folding (`ß` → `ss`), designed for comparison; `lower` is for display and misses such cases.

4. B Superscripts are digits but not decimals; `int("²")` fails. Neither method accepts a sign, so neither is a test for 'parses as an integer'.

5. B Unicode allows the same visible character to be encoded two ways; `unicodedata.normalize("NFC", s)` composes them so equality works.

LESSON 5 · 14 MIN

Parsing input — from lines and tokens to values

Input arrives as text and the program needs values, and everything between is parsing. Most of it is three methods — `split`, `strip`, `partition` — plus `int` and `float` with `ValueError` caught, arranged into a handful of shapes: tokens on a line, delimited fields, `key=value` pairs, a count followed by records, blocks separated by blank lines, and the occasional format that needs a small hand-written scanner. This lesson gives each shape a template, states the validation rule (convert strictly, reject clearly), and shows where the bulk-read idiom `sys.stdin.read().split()` beats reading line by line.

Tokens on a line

```
nums = list(map(int, input().split()))
a, b = map(int, input().split())        # exactly two — ValueError otherwise
first, *rest = input().split()          # at least one
```

`split()` without an argument handles any amount of whitespace and drops the ends. The unpacking forms are also validation: the wrong number of tokens raises `ValueError: not enough values to unpack` or `too many values`, which is the right failure for malformed input.

Delimited fields

```
line = "ada, 36, london"
name, age, city = [f.strip() for f in line.split(",")]
```

`split(",")` keeps empty fields, so `"a,,c"` has three; strip each field, because `", "` after a comma is common. When fields can contain the delimiter (quoted CSV), do not split by hand — the `csv` module (Module 14) handles quoting. A fixed number of fields can be unpacked; a variable number stays a list.

key=value pairs

```
settings = {}
for tok in "host=db port=5432 debug".split():
    key, sep, value = tok.partition("=")
    settings[key] = value if sep else True
```

`partition` splits at the *first* separator and returns three parts, with an empty separator when it is absent — so a `value` containing `=` survives (`"a=b=c"` → `("a", "=", "b=c")`), and a bare flag is detected by `sep` being empty. `split("=", 1)` is the two-element alternative but raises on unpacking when the separator is missing.

A count, then records

```
n = int(input())
rows = []
for _ in range(n):
    name, qty, price = input().split()
    rows.append((name, int(qty), float(price)))
```

Convert as you read so that a bad line fails at the line, not later in the middle of a calculation. When the count line is absent and the records run to the end of input, `for line in sys.stdin` replaces the `range`.

Blocks separated by blank lines

```
import sys
text = sys.stdin.read()
blocks = [b.splitlines() for b in text.strip().split("\n\n") if b.strip()]
```

Read everything, split on the double newline, then split each block into lines. `strip()` on the whole text first avoids a leading or trailing empty block. A variant, one record per block with `key: value` lines, is `dict(line.split(": ", 1) for line in block)`.

Bulk tokens

```
import sys
data = sys.stdin.read().split()
it = iter(data)
n = int(next(it))
values = [int(next(it)) for _ in range(n)]
```

When the input is large — tens of thousands of numbers — reading it in one call and walking an iterator is several times faster than `input()` per line, and it does not care how the numbers are wrapped across lines. `next(it)` raises `StopIteration` if the data runs out, which a wrapper can turn into a clear message. Module 20 uses this as the interview template.

Validating

Convert strictly and catch the failure at the point of conversion:

```
def parse_age(tok):
    try:
        age = int(tok)
    except ValueError:
        raise ValueError(f"age must be an integer, got {tok!r}") from None
    if not 0 ⩽ age ⩽ 150:
        raise ValueError(f"age out of range: {age}")
    return age
```

Two rules. Do not pre-check with `isdigit` — it rejects `-5` and accepts `²`; `int()` is the definition of "parses as an integer". And do not let a bad value through as `0` or `None` silently — either raise, or record the failure and report it (`bad lines: 3`), depending on what the program is for. In a judged exercise the prompt says which; "print `invalid` and continue" is the common contract.

A hand-written scanner

Some formats are neither tokens nor fields: a number followed directly by a unit (`12kg`), an expression without spaces (`3+4*2`), a run-length code (`a3b2`). The tool is an index walked by hand, reading a run of characters of one class at a time:

```python
def scan_number_and_unit(s):
    i = 0
    while i < len(s) and (s[i].isdigit() or s[i] == "."):
        i += 1
    return float(s[:i]), s[i:].strip()

def run_length_decode(code):
    out, i = [], 0
    while i < len(code):
        ch = code[i]
        i += 1
        j = i
        while j < len(code) and code[j].isdigit():
            j += 1
        out.append(ch * int(code[i:j]))
        i = j
    return "".join(out)
```

The shape is always: a position `i`; a loop that advances `i` past one lexical unit; the slice `s[start:i]` as the token. Regular expressions (next lesson) express the same thing declaratively and are the better tool once the format has more than two token kinds.

Pitfalls

`split(" ")` for whitespace tokens — it keeps empties between double spaces.

Forgetting to `strip()` fields after `split(",")`.

`int(line)` where `line` came from `sys.stdin` iteration and still has `\n` — actually fine (`int` tolerates whitespace), but `line == "END"` is not; compare against `line.rstrip("\n")`.

Reading `n` and then reading `n` *tokens* on one line with `input()` per token — `input()` reads a *line*.

Checking `isdigit()` before `int()`.

Silent defaults for unparsable values.

Key takeaways

`split()` for whitespace tokens, `split(sep)` for fields (strip each), `partition` for `key=value`.

Unpacking validates token counts; `int` / `float` validate values — catch `ValueError`, never pre-check with `isdigit`.

Count-then-records with `range(n)`; records to the end with `for line in sys.stdin`; blocks with `read().split("\n\n")`.

`sys.stdin.read().split()` plus an iterator is the fast bulk reader.

A hand scanner is an index advanced past one token class at a time; regexes take over when the grammar grows.

EXERCISE 1

Config parser

Parse a configuration read until the end of input. Blank lines and lines starting with `#` are ignored. Every other line must be `key = value` (spaces around `=` optional) — split with `partition`, and print `invalid: <line>` when there is no `=`. Type each value: an integer, else a float, else `true` / `false` (any case) as a boolean, else a string. Print the entries in input order.

Input: lines. **Output:** `<key>: <type> <value>` per entry (types `int`, `float`, `bool`, `str`; booleans print as `True` / `False`).

```
# server
host = localhost
port=8080
ratio = 0.75
debug = true
name
```

prints

```
host: str localhost
port: int 8080
ratio: float 0.75
debug: bool True
invalid: name
```

INPUT 1

```
# server
host = localhost
port=8080
ratio = 0.75
debug = true
name
```

OUTPUT 1

```
host: str localhost
port: int 8080
ratio: float 0.75
debug: bool True
invalid: name
```

REFERENCE SOLUTION

```
import sys


def typed(text):
    try:
        return "int", int(text)
    except ValueError:
        pass
    try:
        return "float", float(text)
    except ValueError:
        pass
    if text.lower() in ("true", "false"):
        return "bool", text.lower() == "true"
    return "str", text


for line in sys.stdin:
    line = line.strip()
    if not line or line.startswith("#"):
        continue
    key, sep, value = line.partition("=")
    if not sep:
        print(f"invalid: {line}")
        continue
    kind, val = typed(value.strip())
    print(f"{key.strip()}: {kind} {val}")
```

EXERCISE 2

Run-length codec

Each line is `E <text>` (encode) or `D <code>` (decode). Encoding turns runs of a repeated character into the character followed by its count (`aaabcc` → `a3b1c2`). Decoding reverses it; counts may have several digits (`x10`). Write the decoder as a hand scanner: an index that reads one character, then the run of digits after it. Text after the command is taken literally (it may contain spaces).

Input: lines. **Output:** one line per input line (an empty result prints `(empty)`).

```
E aaabcc
D a3b1c2
```

prints

```
a3b1c2
aaabcc
```

INPUT 1

```
E aaabcc
D a3b1c2
```

OUTPUT 1

```
a3b1c2
aaabcc
```

REFERENCE SOLUTION

```python
import sys


def encode(text):
    out = []
    i = 0
    while i < len(text):
        j = i
        while j < len(text) and text[j] == text[i]:
            j += 1
        out.append(f"{text[i]}{j - i}")
        i = j
    return "".join(out)


def decode(code):
    out = []
    i = 0
    while i < len(code):
        ch = code[i]
        i += 1
        j = i
        while j < len(code) and code[j].isdigit():
            j += 1
        out.append(ch * int(code[i:j]))
        i = j
    return "".join(out)


for line in sys.stdin:
    line = line.rstrip("\n")
    cmd, _, rest = line.partition(" ")
    result = encode(rest) if cmd == "E" else decode(rest)
    print(result if result else "(empty)")
```

QUIZ

1. Which reads two integers from one line?

A. `a, b = input()`

B. `a, b = map(int, input().split())`

C. `a = int(input()); b = int(input())`

D. `a, b = int(input().split())`

2. What does `"flag".partition("=")` return?

A. `('flag', '', '')`

B. `('flag',)`

C. `ValueError`

D. `('', '', 'flag')`

3. Why is `if s.isdigit(): n = int(s)` a poor validation?

A. `isdigit` is slow

B. It rejects negative numbers and accepts characters like `²` that `int` rejects; try `int(s)` and catch `ValueError` instead

C. `isdigit` returns a string

D. It is correct

4. What is the fastest way to read a large whitespace-separated input?

A. `input()` in a loop

B. `sys.stdin.read().split()` once, then walk the tokens

C. `sys.stdin.readline()` per token

D. `open(0).readlines()` then `split` each line

5. What does `text.strip().split("\n\n")` give for text made of paragraphs separated by blank lines?

A. A list of paragraphs, each still containing its internal newlines

B. A list of lines

C. A single string

D. A list of words

ANSWER KEY · PARSING INPUT — FROM LINES AND TOKENS TO VALUES

1. B Split the line, convert each token, unpack. The first unpacks characters; the third reads two *lines*; the last calls `int` on a list.

2. A When the separator is absent, the head is the whole string and the other two parts are empty — so testing the separator part tells you whether `=` was present.

3. B `int()` is the definition of 'parses as an integer'; the character test is neither necessary nor sufficient.

4. B One read and one split are a handful of C-level operations; per-line calls pay Python overhead on each line. It also ignores how the tokens are wrapped.

5. A Splitting on the double newline separates blocks; `splitlines()` on each block then gives its lines. Stripping first avoids empty leading or trailing blocks.

LESSON 6 · 15 MIN

Regular expressions — the re module

A regular expression describes a set of strings by a pattern — "one or more digits", "a word, a colon, then anything" — and the `re` module finds, extracts, replaces and splits by such patterns. It is the right tool when the format is more than one delimiter deep, when you need "the numbers anywhere in this text", or when validation is a shape rather than a value. It is the wrong tool for anything `split` or `startswith` can do, and for parsing nested structures. This lesson gives the syntax that covers most uses, the six functions, groups, flags, and the greedy-versus-lazy rule.

The functions

```
import re
text = "order 66 shipped 2024-05-01 to ada"

re.search(r"\d+", text)              # first match anywhere: <re.Match ... mat
re.match(r"\w+", text)               # match only at the *start*: 'order'
re.fullmatch(r"\w+", "order")        # the whole string must match
re.findall(r"\d+", text)             # every match as a list: ['66', '2024',
re.finditer(r"\d+", text)            # every match as Match objects, lazily
re.sub(r"\d", "#", text)             # replace: 'order ## shipped ####-##-## t
re.split(r"[\s-]+", text)            # split on a pattern
```

`search`, `match` and `fullmatch` return a `Match` object or `None` — always test before using it. `m.group()` is the matched text, `m.start()` / `m.end()` its position, `m.group(1)` the first parenthesised group. Patterns are written as raw strings (`r"..."`) so that backslashes reach the regex engine intact.

The syntax

Pattern	Matches
`.`	any character except newline
`\d` `\w` `\s`	digit, word character (letters, digits, `_`), whitespace
`\D` `\W` `\S`	the complements
`[abc]` `[a-z]` `[^0-9]`	a set, a range, a negated set
`^` `$`	start and end of the string (or line, with `re.M`)
`\b`	a word boundary
`x*` `x+` `x?`	zero or more, one or more, zero or one
`x{3}` `x{2,5}` `x{2,}`	exactly, between, at least
`a|b`	alternation
`(...)`	a group: captures, and groups a quantifier
`(?:...)`	a non-capturing group
`(?P<name>...)`	a named group
`(?=...)` `(?!...)`	lookahead: followed by / not followed by, without consuming
`\.` `\(` `\\`	a literal special character

```
re.fullmatch(r"[A-Z]{3}-\d{4}", "ABC-1234")          # a plate
re.search(r"\b(\w+)\s+\1\b", "the the cat")          # a repeated word (\1 =
re.findall(r"[\w.]+@[\w.]+\.\w+", text)              # email-shaped things (
```

Groups

```
m = re.search(r"(\d{4})-(\d{2})-(\d{2})", text)
m.group(0)              # '2024-05-01' — the whole match
m.group(1), m.group(3)  # '2024', '01'
m.groups()              # ('2024', '05', '01')

m = re.search(r"(?P<year>\d{4})-(?P<month>\d{2})", text)
m.group("year")         # '2024'
m.groupdict()           # {'year': '2024', 'month': '05'}
```

`findall` returns the groups as tuples when the pattern has groups (and just group 1 when it has one), which surprises people who wanted the whole match — use `(?:...)` for grouping without capturing, or `finditer` and `m.group()`.

Substitution

```
re.sub(r"(\w+)@(\w+)", r"\2 at \1", "ada@example")        # 'example at ada' -
re.sub(r"\d+", lambda m: str(int(m.group()) * 2), "a1 b22")  # 'a2 b44' — a fu
re.sub(r"\s+", " ", "  many   spaces  ").strip()            # 'many spaces'
```

The replacement string understands `\1` and `\g<name>`; a function receives the `Match` and returns the replacement, which is how a transformation (upper-casing, arithmetic) is applied to each match.

Greedy and lazy

Quantifiers are greedy: they match as much as possible and back off only if the rest of the pattern fails. `<.*>` on `<a><b>` matches the *whole* string. Add `?` to make a quantifier lazy — `<.*?>` matches `<a>` — or, usually better, forbid the closing character in the set: `<[^>]*>`. The negated set is faster and clearer than laziness.

Flags and compiling

```
re.findall(r"^\w+", text, flags=re.M)       # ^ and $ per line
re.search(r"hello", "HELLO", re.I)          # ignore case
re.compile(r"""
    (?P<key>\w+)   # the key
    \s*=\s*
    (?P<val>.*)    # the value
""", re.X)                                  # verbose: whitespace and comments
```

`re.compile(pattern)` returns a pattern object with the same methods (`p.search(s)`); the module functions cache recent patterns, so compiling is about readability and reuse, not speed, except in a tight loop. `re.S` (dotall) lets `.` match newlines.

When not to use a regex

`s.startswith("x")`, `"," in s`, `s.split(",")`, `s.isdigit()` and `s.replace(a, b)` are each clearer and faster than the equivalent regex. Nested or balanced structure (matching brackets, HTML) is beyond regular languages; use a parser. Validation of real-world formats (emails, URLs) with a regex is a rough filter, never a proof. And a regex that took ten minutes to write will take the next reader ten minutes to read — the `re.X` flag with comments is the mitigation.

Pitfalls

- Forgetting the `r` prefix and losing `\b` (backspace) or `\1`.
- Using the result of `search` / `match` without checking for `None`.
- `match` when `search` was meant — `match` anchors at the start.
- `findall` with groups returning tuples instead of whole matches.
- A greedy `.*` spanning far more than intended.
- Special characters unescaped: `.` matches anything, `+` quantifies; `re.escape(s)` escapes a literal.

Key takeaways

- `search` finds anywhere, `match` at the start, `fullmatch` the whole; all return `Match` or `None` — test it.

`findall` / `finditer` for all matches, `sub` (with backreferences or a function) to replace, `split` on a pattern.

Groups capture with `(...)`, name with `(?P<n>...)`, skip capturing with `(?:...)`; `\1` refers back.

Quantifiers are greedy; `?` makes them lazy; a negated set `[^>]*` is usually better than either.

Raw strings always; flags `I`, `M`, `S`, `X`; prefer a string method when one does the job.

EXERCISE 1

Extract the numbers

For each line of text, find every number — an optional minus sign, digits, and optionally a decimal point with more digits — with one `re.findall` pattern that uses a non-capturing group for the fraction. Print the matched texts and their sum (as floats, two decimals).

Input: lines of text. **Output:** `found: <matches>` (or `found: none`) then `sum: <total>`, per line.

```
Order 66 costs 12.50, tax 2.5
no digits here
```

prints

```
found: 66 12.50 2.5
sum: 81.00
found: none
sum: 0.00
```

INPUT 1

```
Order 66 costs 12.50, tax 2.5
no digits here
```

OUTPUT 1

```
found: 66 12.50 2.5
sum: 81.00
found: none
sum: 0.00
```

REFERENCE SOLUTION

```python
import re
import sys

NUMBER = re.compile(r"-?\d+(?:\.\d+)?")

for line in sys.stdin:
    found = NUMBER.findall(line)
    print("found:", " ".join(found) if found else "none")
    print(f"sum: {sum(float(x) for x in found):.2f}")
```

EXERCISE 2

Log lines with named groups

A log line looks like `2024-05-01 12:30:45 [ERROR] Disk full on /dev/sda1`. Write one pattern with named groups `date`, `time`, `level` and `msg`, apply it with `fullmatch`, and print `<level> @ <time>: <msg>` for matching lines and `skip` for the others. After all lines, print the count per level in alphabetical order as `levels: ERROR=1 INFO=2` (or `levels: none`).

Input: lines. **Output:** one line per input line, then the summary.

```
2024-05-01 12:30:45 [ERROR] Disk full on /dev/sda1
2024-05-01 12:31:00 [INFO] Retrying
garbage
```

prints

```
ERROR @ 12:30:45: Disk full on /dev/sda1
INFO @ 12:31:00: Retrying
skip
levels: ERROR=1 INFO=1
```

INPUT 1

```
2024-05-01 12:30:45 [ERROR] Disk full on
/dev/sda1
2024-05-01 12:31:00 [INFO] Retrying
garbage
```

OUTPUT 1

```
ERROR @ 12:30:45: Disk full on /dev/sda1
INFO @ 12:31:00: Retrying
skip
levels: ERROR=1 INFO=1
```

REFERENCE SOLUTION

```python
import re
import sys

LINE = re.compile(r"(?P<date>\d{4}-\d{2}-\d{2}) (?P<time>\d{2}:\d{2}:\d{2}) \[(?P<le

counts = {}
for line in sys.stdin:
    m = LINE.fullmatch(line.rstrip("\n"))
    if m is None:
        print("skip")
        continue
    level = m.group("level")
    print(f"{level} @ {m.group('time')}: {m.group('msg')}")
    counts[level] = counts.get(level, 0) + 1
if counts:
    print("levels:", " ".join(f"{k}={counts[k]}" for k in sorted(counts)))
else:
    print("levels: none")
```

QUIZ

1. What does `re.match(r"\d+", "abc 123")` return?

A. A match for `123`

B. `None` — `match` anchors at the start of the string

C. `['123']`

D. `ValueError`

2. What is `re.findall(r"(\d)-(\d)", "1-2 3-4")`?

A. `['1-2', '3-4']`

B. `[('1', '2'), ('3', '4')]`

C. `['1', '2', '3', '4']`

D. `[<Match>, <Match>]`

3. On `"<a><b>"`, what does `re.search(r"<.*>", s).group()` match?

A. `<a>`

B. `<a><b>` — the quantifier is greedy

C. `<b>`

D. `<>`

4. Why write patterns as raw strings?

A. They run faster

B. So that backslashes reach the regex engine intact — `"\b"` in a normal string is a backspace

C. Raw strings allow `{}` in patterns

D. It is only convention

5. What does `re.sub(r"(\w+) (\w+)", r"\2 \1", "ada lovelace")` return?

A. `'lovelace ada'`

B. `'ada lovelace'`

C. `'\2 \1'`

D. `TypeError`

6. Which task should *not* be done with a regular expression?

A. Finding every date in free text

B. Checking that a string ends with `.txt`

C. Extracting `key=value` pairs from a line

D. Validating a plate format like `ABC-1234`

ANSWER KEY · REGULAR EXPRESSIONS — THE RE MODULE

1. B `match` succeeds only if the pattern matches at position 0. `search` would find `123`.

2. B With capturing groups, `findall` returns tuples of the groups, not the whole matches. Use `(?:...)` or `finditer` for the full text.

3. B `.*` takes as much as it can while the rest still matches. `<.*?>` or, better, `<[^>]*>` matches just `<a>`.

4. B Python's own escape processing would consume or alter sequences like `\b` and `\1`; the `r` prefix turns it off.

5. A Backreferences in the replacement insert the captured groups in the new order.

6. B `s.endswith(".txt")` is clearer and faster; regexes earn their place when the shape is more than a fixed prefix, suffix or delimiter.

CHECKPOINT · 25 MIN

Checkpoint — Strings and text

This checkpoint covers the whole module: strings as immutable sequences of code points, slicing and the method set, f-strings and the format-spec mini-language, characters and the text/bytes boundary with the three digit tests, the parsing shapes for tokens, fields, `key=value` pairs, records and blocks, and regular expressions with groups, substitution and the greedy rule.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What is `"abcdef"[1:5:2]`, and what does `[::-1]` do?
- Why is `"file.txt".strip(".txt")` wrong, and what is right?
- What do `f"{42:06d}"`, `f"{3.14159:.2f}"` and `f"{'ab':>5}"` produce?
- What is the difference between `len("é")` and `len("é".encode())`?
- Which of `isdecimal`, `isdigit`, `isnumeric` accepts `"²"`, and which accepts `"-1"`?
- What does `partition("=")` return when the separator is absent?
- What is the difference between `re.match` and `re.search`, and what do they return when nothing matches?

The three programs are a text-statistics report built from the method set and formatted as a table, a `key=value` configuration parser with typed values and clear rejections, and a log extractor that pulls timestamps, levels and messages out of free text with named groups.

EXERCISE 1

Text statistics

Read all of standard input and report, as a two-column table with labels left-aligned in 14 characters and values right-aligned in 8: `lines` (from `splitlines`), `words` (whitespace tokens), `chars` (every character), `unique` (distinct words after `casefold` and stripping punctuation from both ends; empty results do not count), `longest` (the first longest cleaned word, left-aligned like the labels' column but printed in the value column right-aligned), and `avg length` (mean length of cleaned words, two decimals, `0.00` for none).

Input: any text. **Output:** six table rows.

```
The cat sat. The cat ran!
```

prints

```
lines                1
words                6
chars               26
unique               4
longest            the
avg length        3.00
```

INPUT 1

```
The cat sat. The cat ran!
```

OUTPUT 1

```
lines                1
words                6
chars               26
unique               4
longest            the
avg length        3.00
```

REFERENCE SOLUTION

```python
import string
import sys

data = sys.stdin.read()
words = data.split()
cleaned = [w.casefold().strip(string.punctuation) for w in words]
cleaned = [w for w in cleaned if w]
longest = max(cleaned, key=len) if cleaned else "-"
avg = sum(len(w) for w in cleaned) / len(cleaned) if cleaned else 0.0
print(f"{'lines':<14}{len(data.splitlines()):>8}")
print(f"{'words':<14}{len(words):>8}")
print(f"{'chars':<14}{len(data):>8}")
print(f"{'unique':<14}{len(set(cleaned)):>8}")
print(f"{'longest':<14}{longest:>8}")
print(f"{'avg length':<14}{avg:>8.2f}")
```

EXERCISE 2

Query strings

Parse query strings, one per line: fields separated by `&`, each `key=value` or a bare `key` (a flag). A `+` in a value stands for a space. Type each value as in the config parser (`int`, `float`, `bool` for `true` / `false`, else `str`); a value containing commas is a `list` of strings printed joined with `|`; a bare key is `bool True`. A key seen twice on the same line prints `duplicate: <key>` instead of a second entry. Print a blank-free block per line: the entries in order, then `--`.

Input: lines. **Output:** per line, the entries then `--`.

```
name=ada+lovelace&age=36&tags=a,b,c&admin&age=1
```

prints

```
name: str ada lovelace
age: int 36
tags: list a|b|c
admin: bool True
duplicate: age
--
```

INPUT 1

```
name=ada+lovelace&age=36&tags=a,b,c&admin&age=1
```

OUTPUT 1

```
name: str ada lovelace
age: int 36
tags: list a|b|c
admin: bool True
duplicate: age
--
```

REFERENCE SOLUTION

```
import sys


def typed(text):
    if "," in text:
        return "list", "|".join(text.split(","))
    try:
        return "int", int(text)
    except ValueError:
        pass
    try:
        return "float", float(text)
    except ValueError:
        pass
    if text.lower() in ("true", "false"):
        return "bool", text.lower() == "true"
    return "str", text


for line in sys.stdin:
    seen = set()
    for field in line.strip().split("&"):
        if not field:
            continue
        key, sep, value = field.partition("=")
        if key in seen:
            print(f"duplicate: {key}")
            continue
        seen.add(key)
        if not sep:
            print(f"{key}: bool True")
            continue
        kind, val = typed(value.replace("+", " "))
        print(f"{key}: {kind} {val}")
    print("--")
```

EXERCISE 3

Access-log summary

Each line of an access log is `<ip> - - [<timestamp>] "<method> <path> HTTP/1.1" <status> <bytes>`. With one regular expression using named groups, parse every line (skip lines that do not match) and print: the number of requests per status code in ascending numeric order, the total bytes, and the most requested path (ties broken alphabetically).

Input: lines. **Output:** `status <code>: <count>` lines, then `bytes: <total>`, then `top: <path>` (or `top: none`).

```
10.0.0.1 - - [01/May/2024:12:00:00 +0000] "GET /index.html HTTP/1.1" 200 512
10.0.0.2 - - [01/May/2024:12:00:01 +0000] "GET /missing HTTP/1.1" 404 0
10.0.0.1 - - [01/May/2024:12:00:02 +0000] "POST /index.html HTTP/1.1" 200 128
not a log line
```

prints

```
status 200: 2
status 404: 1
bytes: 640
top: /index.html
```

INPUT 1

```
10.0.0.1 - - [01/May/2024:12:00:00
+0000] "GET /index.html HTTP/1.1" 200
512
10.0.0.2 - - [01/May/2024:12:00:01
+0000] "GET /missing HTTP/1.1" 404 0
10.0.0.1 - - [01/May/2024:12:00:02
+0000] "POST /index.html HTTP/1.1" 200
128
not a log line
```

OUTPUT 1

```
status 200: 2
status 404: 1
bytes: 640
top: /index.html
```

REFERENCE SOLUTION

```python
import re
import sys

LOG = re.compile(
    r'(?P<ip>\S+) - - \[(?P<ts>[^\]]+)\] "(?P<method>[A-Z]+) (?P<path>\S+) HTTP/1\.1
)

statuses = {}
paths = {}
total = 0
for line in sys.stdin:
    m = LOG.fullmatch(line.rstrip("\n"))
    if not m:
        continue
    status = int(m.group("status"))
    statuses[status] = statuses.get(status, 0) + 1
    path = m.group("path")
    paths[path] = paths.get(path, 0) + 1
    total += int(m.group("bytes"))

for status in sorted(statuses):
    print(f"status {status}: {statuses[status]}")
print(f"bytes: {total}")
if paths:
    top = min(paths, key=lambda p: (-paths[p], p))
    print(f"top: {top}")
else:
    print("top: none")
```

CHECKPOINT QUESTIONS

1. What is `"hello"[::-1]`?

A. `'olleh'`

B. `'hello'`

C. `'h'`

D. `IndexError`

2. What does `"a,b,,c".split(",")` return?

A. `['a', 'b', 'c']`

B. `['a', 'b', '', 'c']`

C. `['a', 'b', ',', 'c']`

D. `['a,b,,c']`

3. What does `f"{3.14159:8.2f}|"` produce?

A. `'3.14 |'`

B. `' 3.14|'`

C. `'3.14|'`

D. `'00003.14|'`

4. Which expression gives the number of bytes in `s` as UTF-8?

A. `len(s)`

B. `len(s.encode("utf-8"))`

C. `s.__sizeof__()`

D. `len(bytes(s))`

5. What is `"ß".upper()` ?

A. `'ß'`

B. `'SS'`

C. `'ß'`

D. `UnicodeError`

6. What does `"k=v".split("=", 1)` return, and what happens on `"k"`?

A. `['k', 'v']`; `['k']`

B. `('k', 'v')`; `('k',)`

C. `['k', 'v']`; `ValueError`

D. `['k', '=', 'v']`; `['k']`

7. What does `re.search(r"\d+", "abc")` return?

A. `''`

B. `None`

C. `[]`

D. It raises `ValueError`

8. Which pattern matches a whole line that is exactly three uppercase letters?

A. `re.search(r"[A-Z]{3}", s)`

B. `re.fullmatch(r"[A-Z]{3}", s)`

C. `re.match(r"[A-Z]{3}", s)`

D. `re.findall(r"[A-Z]", s)`

9. What does `"x".join(["a", "b", "c"])` produce?

A. `'axbxc'`

B. `'xaxbxcx'`

C. `'abc'`

D. `'a b c'`

10. What is `"Hello World".title().swapcase()`?

A. `'hELLO wORLD'`

B. `'HELLO WORLD'`

C. `'hello world'`

D. `'Hello World'`

11. What does `re.sub(r"\s+", " ", " a b ").strip()` return?

A. `'a b'`

B. `' a b '`

C. `'ab'`

D. `' a b '`

12. Which is true of `"²".isdecimal()`?

A. `True`

B. `False`

C. It raises `UnicodeError`

D. It depends on the locale

ANSWER KEY · CHECKPOINT — STRINGS AND TEXT

1. A A step of −1 walks from the end to the start — the reversal idiom.

2. B An explicit separator keeps empty fields; only `split()` with no argument collapses runs.

3. B Numbers right-align in the width by default; two decimals in a field of eight gives four leading spaces.

4. B `len(s)` counts code points; encoding produces the bytes. `bytes(s)` without an encoding is a `TypeError`.

5. B Case mapping can change the length of a string; the German sharp s uppercases to two letters. `casefold` gives `'ss'` for comparison.

6. A `split` with a maxsplit returns a list of at most two parts; with no separator present the list has one element, so unpacking into two names would fail — which is why `partition` is often safer.

7. B No match gives `None`, which is why the result must be tested before calling `.group()`.

8. B `fullmatch` requires the pattern to cover the entire string; `match` only anchors the start, and `search` anywhere.

9. A `join` places the separator between elements, never at the ends.

10. A `title` capitalises each word; `swapcase` then flips every letter's case.

11. A Runs of whitespace collapse to single spaces; `strip` removes the ends. The same result as `" ".join(s.split())`.

12. B `isdecimal` accepts only characters usable to form base-10 numbers with `int()`; a superscript is a digit (`isdigit` is true) but not a decimal.