PYTHON LESSON

# Iterating Through Strings Exercises

# 💻 Exercises: Iterating Through Strings with a For Loop

*Ten exercises covering all three iteration styles from this lesson, character filtering, accumulator counting, and both duplicate-letter approaches. Exercise 1 is a real practice problem straight from class — the rest build outward from it.*

### Exercise 1: Count Letters Per Word in a Paragraph

Given a `text` variable holding a couple of paragraphs of Lorem Ipsum placeholder text, print every word alongside how many **letters** it contains — numbers and punctuation don't count. For example, a word like `"h3ll!"` should count as **3**: the `h` and both `l`'s are letters, the `3` and `!` aren't, and duplicated letters (like the two `l`'s) are counted separately, not merged into one.

**Goal:** Split the paragraph into words, then run a nested `for` loop — one word at a time, one character at a time within each word — using `.isalpha()` to decide what counts.

→ Solution

### Exercise 2: Reverse a String, Three Ways

Given `word = "programming"`, write three separate solutions that each produce `"gnimmargorp"` — one using `for ch in word:`, one using `for i in range(len(word)):`, and one using `for i, ch in enumerate(word):`. Don't use slicing (`word[::-1]`) or `reversed()` for any of them.

**Goal:** Practise all three iteration forms from this lesson on the exact same problem, so the differences between them are concrete rather than abstract.

→ Solution

### Exercise 3: Vowel vs. Consonant Counter

Given a sentence entered via `input()`, count how many letters are vowels (`a, e, i, o, u` — treat upper and lowercase the same) versus consonants, ignoring anything that isn't a letter at all.

**Goal:** Combine `.isalpha()` filtering with a second classification inside the same loop — not every letter needs its own separate pass.

→ Solution

### Exercise 4: A Fresh Character-Type Accumulator

Given `text = "Order #4521 shipped on 03/15 - contact support@example.com!"`, build the same four-way accumulator (letters, numbers, spaces, special) from this lesson's own case study, applied to this new, messier string.

**Goal:** Confirm the `if` / `elif` / `elif` / `else` pattern still works cleanly on a string with a genuinely wide mix of character types, not just a tidy example sentence.

→ Solution

### Exercise 5: Duplicate Letters — the Two-List Way

Given `word = "bookkeeper"`, use the explicit two-list approach from this lesson (`my_list` and `duplicates`) to print every letter that appears more than once, each one printed only a single time.

**Goal:** Reproduce the lesson's own case study on a new word with a genuinely tricky repeated-letter pattern (`oo`, `kk`, `ee`), confirming the `elif` guard still prevents a triple-listed duplicate.

→ Solution

### Exercise 6: Duplicate Letters — the Shorter Pythonic Way

Solve Exercise 5 again, this time using the single-list `.count()` approach from this lesson, on the same `word = "bookkeeper"`. Confirm both versions produce an identical result.

**Goal:** Practise both taught approaches to the same problem, and see for yourself that "shorter" and "different result" aren't the same thing — the two should always agree.

→ Solution

### Exercise 7: A Case-Insensitive Unique-Character String

Given `word = "Mississippi River"`, build a string containing every distinct character exactly once, treating uppercase and lowercase letters as the same character — applying this lesson's own case-sensitivity fix.

**Goal:** Apply the "normalise both the check and the value you store" rule from this lesson to a fresh string, rather than just reproducing the lesson's own worked example.

→ Solution

### Exercise 8: Manual Word Count Without .split()

Given `sentence = "The quick brown fox jumps over the lazy dog"`, count how many words it contains **without** using `.split()` — walk the string character by character instead, watching for the transition from a space to a non-space character.

**Goal:** Practise detecting a boundary condition (the start of a new word) using only single-character checks, rather than reaching for a built-in that solves the whole problem in one call.

→ Solution

### Exercise 9: Manual Title Case

Given `sentence = "the quick brown fox"`, produce `"The Quick Brown Fox"` using `for i, ch in enumerate(sentence):` — capitalising a character only when it's the first letter of the string, or the first letter right after a space. Don't use `.title()` or `.capitalize()`.

**Goal:** Practise using the index from `enumerate()` for something genuinely useful — checking what character sits immediately before the current one.

→ Solution

### Exercise 10: Palindrome Checker, Index-Based

Write a function that takes a string and returns `True` if it reads the same forwards and backwards, and `False` otherwise — using `for i in range(len(string)):` to compare `string[i]` against a character counted in from the opposite end. Test it against `"level"` and `"python"`.

**Goal:** Practise the "two positions moving toward each other" pattern using pure index arithmetic — a shape that `for ch in string:` alone genuinely can't express, since it never gives you a second position to compare against.

→ Solution