PYTHON LESSON

Iterating Through Strings

Iterating Through Strings with a For Loop

A string is iterable in exactly the same for-each sense as a list — Python doesn't care that its items happen to be single characters instead of list elements. This lesson covers the three real ways to walk a string, how to sort characters into categories (letters, numbers, spaces, everything else), and a full worked case study — spotting duplicate letters — that shows how one throwaway line of Python often replaces several lines of loop bookkeeping in other languages.

> **COMING FROM JAVA / JAVASCRIPT**
>
> **Java strings aren't directly iterable at all** — you either call `.toCharArray()` first to get a `char[]` you can use in an enhanced `for` loop, or index in with `.charAt(i)` inside a classic counting loop. JavaScript strings *are* directly iterable (`for (const ch of str)`), much closer to Python's `for ch in string` — but JS also still supports classic `str[i]` indexing. Python gives you both of those same two shapes without ever needing a conversion step first.

Three Ways to Walk a String

Same underlying `for` loop each time — the difference is what it hands you on every pass.

DIRECT CHARACTER ACCESS

`for ch in string:`

The simplest and most common form — `ch` is a new variable holding the actual character each time, not a position. Reach for this whenever you only care about the character itself, not where it sits.

```
for ch in "Hi!":
    print(ch)
# H
# i
# !
```

INDEX-BASED ACCESS

`for i in range(len(string)):`

`i` is a position (an integer), and `string[i]` gets you the character sitting there. Reach for this when you genuinely need the index itself — comparing a character against its neighbour, for instance.

```
for i in range(len("Hi!")):
    print(i, "Hi!"[i])
# 0 H
# 1 i
# 2 !
```

INDEX AND CHARACTER TOGETHER

THE MORE IDIOMATIC VERSION

`for i in enumerate(string):`

`enumerate()` hands back a tuple of `(index, character)` on every pass. Reading `i[0]` and `i[1]` works — a tuple supports indexing like anything else — but unpacking directly into two variables is the more idiomatic form.

```
for i in enumerate("Hi!"):
    print(i[0], i[1])
# 0 H
# 1 i
# 2 !
```

`for i, ch in enumerate(string):`

Unpacking the tuple straight into two loop variables reads more clearly than indexing into `i[0]`/`i[1]` afterward — both do exactly the same thing, so this is purely a style upgrade, not a correction.

```
for i, ch in enumerate("Hi!"):
    print(i, ch)
# 0 H
# 1 i
# 2 !
```

🔎 Filtering Characters: isalpha() and isnumeric()

Every character can answer a few basic questions about itself. These two come up constantly once you're walking a string one character at a time.

METHOD	RETURNS TRUE FOR
`ch.isalpha()`	Letters only — a–z, A–Z, and other alphabetic Unicode characters
`ch.isnumeric()`	Digit characters
`ch.isspace()`	Whitespace — space, tab, newline

🚀 FASTAPI RELEVANCE

Checking that user-submitted text only contains the characters it's supposed to — a username, a postcode, a product code — is a genuinely common thing to validate before it ever reaches a database. A `for` loop scanning for `.isalpha()`/`.isnumeric()` the way this lesson does is a perfectly reasonable way to write that check by hand, before ever reaching for a regular expression.

🧮 Counting Letters, Numbers, Spaces & Special Characters

Python doesn't hand you a built-in for "how many special characters are in this string" the way it does for letters and digits — so you build the count yourself with a set of accumulator variables and an `if/elif/else` chain.

Sorting Every Character Into a Bucket

Four accumulators, each starting at zero, each incremented in exactly one branch per character.

```
text = "Hello World 123!"

letters = 0
numbers = 0
spaces = 0
special = 0

for ch in text:
    if ch.isalpha():
        letters += 1
    elif ch.isnumeric():
        numbers += 1
    elif ch.isspace():
        spaces += 1
    else:
        special += 1

print(f"Letters: {letters}")
print(f"Numbers: {numbers}")
print(f"Spaces: {spaces}")
print(f"Special: {special}")
# Letters: 10
# Numbers: 3
# Spaces: 2
# Special: 1
```

The `else` branch is doing real work here — there's no `.isspecial()` method to check against directly, so "everything that wasn't a letter, a number, or a space" is the only way to catch punctuation and symbols at all.

> **A style note, not a rule:** *those four `print()` calls could just as easily be written as one longer line, broken across the screen with a backslash continuation, or as a single multi-line f-string. Neither approach is more "correct" than four separate `print()` calls — it's purely a readability preference, and you'll see both in real code.*

🔍 Case Study: Finding Duplicate Letters

A genuinely useful little program — given a string, find every letter that appears more than once, printing each duplicate exactly once even if it repeats three or four times.

The Explicit Two-List Approach

Track everything you've seen in one list, and everything you've already flagged as a duplicate in a second list — the "show your work" version.

```
word = "mississippi"

my_list = []
duplicates = []

for ch in word:
    if ch not in my_list:
        my_list.append(ch)
    elif ch not in duplicates:
        duplicates.append(ch)

print(duplicates)
# ['i', 's', 'p']
```

The `elif` matters here — it only runs when `ch` *is* already in `my_list`, and even then only adds it to `duplicates` if it isn't already flagged. Without that second check, a letter repeated three or four times (like the middle `s`'s in `"mississippi"`) would get added to `duplicates` more than once.

The Shorter Pythonic Version

Same result, using `.count()` instead of a second list.

```
word = "mississippi"

duplicates = []

for ch in word:
    if ch not in duplicates and word.count(ch) > 1:
        duplicates.append(ch)

print(duplicates)
# ['i', 's', 'p']
```

🔬 "SHORTER" ISN'T QUITE THE SAME AS "FASTER"

Both versions do more work than a single pass — but not in the same way. `word.count(ch)` re-scans the *entire original string* on every single character, so its total work grows with the length of `word` every time. The two-list version's checks only scan against `my_list`/`duplicates`, and for real text those stay small — English only has 26 letters, so those lists can never grow past a couple of dozen entries no matter how long `word` gets. For a short word like `"mississippi"` the difference is invisible; for a full paragraph, the two-list version genuinely does less total work, even though it's the longer piece of code.

🧩 Building a Unique-Character String, and a Case-Sensitivity Trap

A closely related task — instead of finding which characters repeat, build a new string containing every character that appears, each one exactly once.

The Case-Sensitivity Trap

This looks right, and works fine for an all-lowercase or all-uppercase string — but watch what happens with mixed case.

```
word = "Hello hello"

my_list = []
```

```
for ch in word:
    if ch not in my_list:
        my_list.append(ch)

unique = "".join(my_list)
print(unique)
# Helo h        <- six entries, not five — "H" and "h" both got through
```

🔬 **WHY "H" AND "H" BOTH SLIP THROUGH**

Python string comparison is case-sensitive by default — `"H" == "h"` evaluates to `False`, so the `ch not in my_list` check genuinely doesn't see them as the same character. From the computer's point of view that's entirely correct: they're two different characters. The bug only exists relative to what the program is *supposed* to do — treat `"H"` and `"h"` as the same letter — which the code as written never actually says.

The Fix — Normalise Before You Check

Convert each character to lowercase before both the membership check and the append — not just one or the other.

```
word = "Hello hello"

my_list = []

for ch in word:
    lower_ch = ch.lower()
    if lower_ch not in my_list:
        my_list.append(lower_ch)

unique = "".join(my_list)
print(unique)
# helo          <- five unique characters: h, e, l, o, and the space
```

Normalise both sides, or neither: *a common half-fix is normalising only the check (`if ch.lower() not in my_list`) but still appending the original `ch`. That still leaves mixed-case originals*

sitting in `my_list`, which quietly breaks the very check you just fixed the next time a repeated letter turns up in the opposite case. The check and the stored value both need to agree on case.

📋 Quick Reference — Iterating Through Strings

TASK	CODE	NOTES
Walk each character	`for ch in string:`	*Most common form — use when you don't need the position*
Walk each index	`for i in range(len(string)):`	`string[i]` *gets the character at position* `i`
Walk index and character together	`for i, ch in enumerate(string):`	*More idiomatic than reading* `i[0]` / `i[1]` *off the raw tuple*
Check if a character is a letter	`ch.isalpha()`	
Check if a character is a digit	`ch.isnumeric()`	
Check if a character is whitespace	`ch.isspace()`	
Count occurrences of a character in a string	`string.count(ch)`	*Re-scans the whole string every call — costly inside a loop over a long string*
Join a list of characters back into a string	`"".join(my_list)`	*The empty string is the "glue" placed between each item*
Normalise case before comparing	`ch.lower()`	*Apply to both the check and the value you store*

⚠ Gotchas for Java / JavaScript Programmers:

Reaching for an index loop out of habit — `for i in range(len(string)):` *is the direct translation of a Java/C-style counting loop, and it always works, but if all you're going to do with* `i` *is immediately look up* `string[i]`, `for ch in string:` *says the same thing with less bookkeeping. Reach for the index form only when you genuinely need the position itself.*

Assigning to a string index throws in Python, but fails silently in JS — *in JavaScript, `str[0] = "x"` quietly does nothing at all; no error, no change, the string is just unaffected. Python is stricter about it: `string[0] = "x"` raises `TypeError: 'str' object does not support item assignment` immediately. If you're used to JS's silent no-op, the loud Python error is actually the more honest behaviour — it tells you straight away that strings are immutable, rather than letting the mistake slip past unnoticed.*

`enumerate()` gives you a tuple, not two separate loop variables automatically — *`for i in enumerate(string):` compiles and runs fine, with `i` holding a two-item tuple you then index into as `i[0]`/`i[1]`. It's correct, but `for i, ch in enumerate(string):` unpacks the same tuple directly into two names and is what you'll see in almost all real Python code.*

`word.count(ch)` inside a loop over `word` is easy to reach for without noticing the cost — *it reads cleanly, but every call re-scans the whole string from the start. On a short word the cost is invisible; on a long paragraph, called once per character, it adds up fast. The two-list approach earlier in this lesson avoids that by only ever scanning against a list of already-seen characters, which stays small.*