PYTHON LESSON

Displaying Output

🖨️ Displaying Output on the Screen

Every language needs a way to show something to the user, and Python's answer is a single, deceptively simple function: `print()`. What makes it interesting isn't the function itself — it's the handful of different ways Python lets you build the text you hand to it. String concatenation, f-strings, and `.format()` all solve the same problem, but they weren't all designed at the same time, and code you'll read in the wild uses all three. Knowing which one to reach for — and why f-strings won — is the real subject of this lesson.

> ⚡ COMING FROM JAVA / JAVASCRIPT
>
> **There's no `System.out.println()` and no `console.log()` — just `print()`**, and it already does more than either of those by default: it accepts *any number* of comma-separated arguments, automatically converts each one to a string, joins them with a space, and appends a newline — all without you asking for any of it. Java needs explicit `+` concatenation or `String.format()`; JavaScript needs template literals (`` `${x}` ``) or comma-separated `console.log` arguments (which — unlike Python — don't auto-stringify objects into readable text nearly as often). Python's f-strings, introduced in 3.6, are the closest cousin to JavaScript's template literals — same `${expr}`-style idea, different bracket (`{expr}`) and a mandatory `f` prefix.

🧱 The Building Blocks

`print()` is one function, but it has more going on under the hood than its name suggests — and there are three genuinely different ways to assemble the string you pass to it.

MULTIPLE ARGUMENTS

`print(a, b, c)`

Pass as many comma-separated values as you like. Python converts each one to a string automatically (no manual `str()` needed) and joins them with a single space by default.

```
print("Score:", 42, "/", 50)
# → Score: 42 / 50
```

SEP AND END

`print(a, b, sep=..., end=...)`

`sep` replaces the default single-space joiner between arguments. `end` replaces the default trailing newline — set it to `""` to keep printing on the same line.

```
print("a", "b", "c", sep="-")   # a-b-c
print("Loading", end="")
print("...")   # Loading...
```

STRING CONCATENATION

`"a" + str(n) + "b"`

The oldest approach: glue strings together with `+`. Every non-string value needs an explicit `str()` call first — Python won't silently convert a number for you the way `print()`'s own comma-separated arguments do.

```
name = "Ana"
age = 30
print("Name: " + name + ", Age: " + str(age))
```

F-STRINGS (3.6+)

`f"Name: {name}"`

Prefix the string with `f`, then drop any expression straight into `{ }` — no concatenation, no manual `str()`. This is the modern, preferred way to build formatted strings in Python.

```
name = "Ana"
age = 30
print(f"Name: {name}, Age: {age}")
```

.FORMAT() METHOD

`"{} is {}".format(a, b)`

The predecessor to f-strings (Python 2.6+): placeholders written as `{}` or `{0}`/`{1}` get filled in, in order, by `.format()`'s arguments. Still common in older codebases and library documentation.

```
print("Name: {}, Age: {}".format(name, age))
print("{1} is {0}".format(age, name))   # reorder by index
```

FORMAT SPECIFIERS

`f"{value:.2f}"`

A colon inside the `{ }` introduces a mini-language for controlling exactly how a value is displayed — decimal places, thousands separators, minimum width, and left/right/centre alignment.

```
price = 19.9
print(f"£{price:.2f}")      # £19.90
print(f"{7:05d}")           # 00007
print(f"{'hi':>10}")        # right-align in 10 chars
```

📖 Why Concatenation Falls Apart Fast

A single `+` is harmless. The problem shows up once a line needs several values mixed with text — concatenation forces every non-string value through `str()` by hand, and every join point needs its own `+`.

The Same Line, Three Ways

Watch how the concatenation version grows noisier as more values are added, while the f-string version barely changes shape at all.

```
item = "Notebook"
qty = 3
price = 4.5
total = qty * price

# Concatenation — every value needs its own str() and its own +
print(item + " x" + str(qty) + " = £" + str(total))

# .format() — placeholders filled in argument order
print("{} x{} = £{}".format(item, qty, total))

# f-string — the value sits right where it's used, no str() needed
print(f"{item} x{qty} = £{total}")

# All three print: Notebook x3 = £13.5
```

🗒️ Case Study: Printing an Aligned Receipt

Format specifiers earn their keep once output needs to actually line up — a plain f-string with no specifier leaves every price at a different width, which looks fine for one line and wrong the moment there's a second line to compare it against.

From Class: A Three-Item Receipt, Right-Aligned

`{name:<10}` left-aligns `name` inside a 10-character field; `{price:>8.2f}` right-aligns `price` inside an 8-character field, always showing exactly two decimal places.

```
items = [("Notebook", 4.5), ("Pen", 1.2), ("Eraser", 0.75)]

total = 0
for name, price in items:
    print(f"{name:<10} £{price:>8.2f}")
    total += price

print(f"{'-' * 19}")
print(f"{'Total':<10} £{total:>8.2f}")

# Notebook   £    4.50
# Pen        £    1.20
# Eraser     £    0.75
# -------------------
# Total      £    6.45
```

📋 Quick Reference — Displaying Output

TASK	SYNTAX	NOTES
Print several values on one line	`print(a, b, c)`	*Auto-converts to string, joins with a space by default*
Change the joiner between arguments	`print(a, b, sep="-")`	*Default separator is a single space*
Suppress the trailing newline	`print(x, end="")`	*Default `end` is `"\n"`*
Concatenate strings and values	`"a" + str(n)`	*Every non-string needs an explicit `str()`*
Embed an expression in a string	`f"{expr}"`	*Preferred since Python 3.6 — clearest, fastest to write*
Fill positional placeholders	`"{} {}".format(a, b)`	*Common in code written before f-strings existed*
Fixed decimal places	`f"{x:.2f}"`	*Rounds, doesn't truncate*
Left / right / centre align	`f"{x:<10}"` / `f"{x:>10}"` / `f"{x:^10}"`	*Number after the symbol is the field width*
Zero-pad an integer	`f"{n:05d}"`	*Pads with leading zeros to 5 total digits*
Debug a variable quickly	`f"{x=}"`	*3.8+ — prints both the expression text and its value, e.g. `x=42`*

⚠ Gotchas for Java / JavaScript Programmers:

Concatenating a number with `+` raises `TypeError`, it doesn't silently coerce — *`"Age: " + 30` crashes in Python, whereas Java happily turns `"Age: " + 30` into `"Age: 30"` and JavaScript does the same. Python requires `str(30)` first, or (better) an f-string, which sidesteps the whole issue.*

An f-string still needs the `f` prefix — *`"{name}"` without the leading `f` prints the literal text `{name}`, curly braces and all, instead of raising an error. This is a silent bug, not a crash, so it's easy to miss.*

`print()`'s default `end` is a newline, not nothing — *unlike JavaScript's `process.stdout.write()` or manually building a string in Java, every bare `print()` call adds its own line break. Forgetting this is why a loop of `print(x, end="")` calls without a final newline can leave your very next output glued onto the same line.*

Format-spec syntax is its own mini-language, not Python expressions — *`{x:>10}` means "right-align in a 10-char field," not "compare `x` to 10." It only appears after the colon inside `{ }`, and it isn't valid anywhere else in Python.*