PYTHON LESSON

For Loops And Range

🔁 For Loops & the range() Function

Python has exactly one `for` loop, and it only knows how to do one thing: walk through the items of something iterable, one at a time, until there aren't any left. There's no separate "counting" version. When you want to loop a specific number of times — the thing a C-style `for (int i = 0; i < n; i++)` does in Java or JavaScript — Python doesn't give the loop a second mode. Instead, it gives you `range()`: a tool that produces something to iterate over, so the exact same single `for` loop can handle both jobs.

> **COMING FROM JAVA / JAVASCRIPT**
>
> **Java and JavaScript actually have two different kinds of `for` loop** — the C-style counting form (`for (int i = 0; i < n; i++)`) and the for-each form (`for (String s : list)` in Java, `for (const x of list)` in JS). Python only has the for-each form. There is no Python equivalent of the three-clause C-style loop at all — `range()` exists specifically so that "loop `n` times" can still be expressed using the one loop Python actually has.

🧱 The For Loop Itself

Before `range()` even enters the picture, a `for` loop just walks whatever you hand it.

BASIC FORM

`for item in sequence:`

`item` is a new variable, created by the loop itself, that takes on each value in `sequence` in turn. No indexing, no manual counter.

```
for name in ["Philip", "Alex", "Sam"]:
    print(name)
```

WORKS ON ANYTHING ITERABLE

`strings, lists, dicts, ranges...`

A string iterates character by character. A dict iterates its keys by default. Anything that knows how to hand out its items one at a time can go after `in`.

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

📏 range() — The Three Forms

`range()` takes one, two, or three arguments, and what it means changes depending on how many you give it.

ONE ARGUMENT

`range(stop)`

Counts from `0` up to, but **not including**, `stop`. `range(5)` produces 5 values: `0, 1, 2, 3, 4` — never `5` itself.

```
for i in range(5):
    print(i)   # 0 1 2 3 4
```

TWO ARGUMENTS

`range(start, stop)`

Counts from `start` up to, but not including, `stop`. `range(1, 6)` produces `1, 2, 3, 4, 5` — five values, not six.

```
for i in range(1, 6):
    print(i)   # 1 2 3 4 5
```

THREE ARGUMENTS

`range(start, stop, step)`

Same as above, but counts in increments of `step` instead of 1. `step` can be negative, which is how you count downward.

```
for i in range(0, 10, 2):
    print(i)   # 0 2 4 6 8

for i in range(5, 0, -1):
    print(i)   # 5 4 3 2 1
```

`STOP` IS ALWAYS EXCLUSIVE

`range(1, 11)` → 1 through 10

This is the single most common off-by-one trap with `range()`. If you want to count 1 through 10 *inclusive*, the second argument has to be `11`, not `10`.

```
list(range(1, 11))
# [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

🚀 **FASTAPI RELEVANCE**

Pagination is the most common place `range()`-style counting shows up in real API code — generating an offset for a page of results (`range(0, total, page_size)`), or looping a fixed number of retry attempts against an external service. Nothing exotic; it's the same counting loop from this lesson, just wired into a request handler.

🔬 Why `print(range(1, 11))` Doesn't Print a List

This genuinely surprises a lot of people the first time they hit it, and it's worth understanding properly rather than just memorising the workaround.

The Behaviour Itself

```
print(range(1, 11))
# range(1, 11)          ← not the numbers!

print(list(range(1, 11)))
# [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]   ← now it's the numbers
```

✅ VERIFIED: RANGE() IS A LAZY OBJECT, NOT A CONTAINER OF NUMBERS

`range(1, 11)` doesn't return a list at all — it returns an object of type `range`, which only stores its `start`, `stop`, and `step`. It computes each value on demand as the `for` loop asks for the next one; it never builds the full sequence of numbers in memory up front. `print()` shows the object's own representation — `range(1, 11)` — the same way it would show `<something>` for any other object that hasn't been converted into something printable.

`list(range(1, 11))` explicitly walks the range object and collects every value it produces into a real, fully-built list — *that's* what finally gets printed as `[1, 2, ..., 10]`.

```
import sys

print(sys.getsizeof(range(1, 11)))          # 48 bytes
print(sys.getsizeof(range(1, 10_000_000))) # 48 bytes — identical!
print(sys.getsizeof(list(range(1, 11))))    # 136 bytes
```

Those numbers are real — measured directly, not estimated. A range covering 10 million numbers takes exactly as much memory as one covering ten, because it never actually stores the numbers — only the three integers needed to generate any of them on request. A materialised list of 10 million numbers would take megabytes.

A little history, for context: *this wasn't always how it worked. In Python 2, `range()` really did build and return a full list immediately — and a separate function, `xrange()`, existed specifically to provide this same lazy, memory-efficient behaviour when you didn't want the whole list built up front. Python 3 removed that split: `range()` was redefined to always behave the way `xrange()` used to, and the old list-building behaviour was dropped entirely. If you ever see `xrange()` in someone's code, that's a sure sign it's Python 2.*

📋 Quick Reference — For Loops & range()

TASK	CODE	NOTES
Loop over a list	`for x in my_list:`	*Works on any iterable — lists, tuples, strings, dict keys, sets*
Loop 5 times	`for i in range(5):`	*Produces 0, 1, 2, 3, 4 — five values*
Loop from 1 to 10 inclusive	`for i in range(1, 11):`	`stop` *is exclusive, so it has to be one past the last value you want*
Loop in steps of 2	`for i in range(0, 10, 2):`	*0, 2, 4, 6, 8*
Count downward	`for i in range(5, 0, -1):`	*5, 4, 3, 2, 1 — needs a negative step*
Turn a range into an actual list	`list(range(1, 11))`	*Only do this if you actually need the values stored — a* `for` *loop never needs this step*
Loop over both index and value	`for i, x in enumerate(my_list):`	*Covered in a future lesson — mentioned here since it often replaces* `range(len(my_list))`

⚠ Gotchas for Java / JavaScript Programmers:

Python has one for loop, not two — *there is no C-style* `for (init; condition; increment)` *in Python at all.* `range()` *is what lets the single for-each loop cover the "count from A to B" case too.*

`range()` **is lazy —** `lodash`**'s** `_.range()` **is not** — *if you're used to JavaScript's* `lodash` *library, its* `_.range(1, 11)` *immediately builds and returns a real array. Python's* `range()` *deliberately does the opposite: it builds nothing until something actually asks it for values. Don't assume the two behave the same way just because the name and arguments look similar.*

The loop variable isn't block-scoped — *in Java, a variable declared in a* `for` *loop's header doesn't exist outside the loop. In Python, the loop variable (*`i`*,* `name`*, whatever you called it) is a completely ordinary variable that keeps its last value and stays accessible after the loop finishes. Relying on this is generally poor style, but it will not raise an error the way it would in Java.*

`stop` **is exclusive, same trap as JS's** `Array.slice()` — *if you've been burned by* `slice(1, 5)` *not*

> *including index 5 in JavaScript, it's the exact same "goes up to, not including" convention here. Consistent, at least, once you know to expect it.*