PYTHON LESSON

For Vs While Loops

⚖️ For vs While Loops — A Deeper Comparison

This is a follow-up to the previous lesson on `for` loops and `range()` — worth reading first if you haven't. Here we go further: comparing `for` and `while` loops directly for the same job, actually measuring which one runs faster rather than guessing, and looking at the one loop construct Python deliberately doesn't have — `do...while`.

> ⚡ COMING FROM JAVA / JAVASCRIPT
>
> **Both loops exist in Java and JavaScript too**, and the basic distinction carries over directly: a `for` loop is the natural choice when you're working through a known collection or a known number of repetitions; a `while` loop is the natural choice when you're repeating "until some condition changes," with no fixed number of iterations decided in advance. What's different in Python is covered below — no C-style `for`, and no `do...while` at all.

✍️ The Same Task, Two Ways

Take a simple task — print "Philip" five times — and write it both ways.

While Loop: Four Lines, an Explicit Counter

```
index = 0
while index < 5:
    print("Philip")
    index += 1
```

For Loop: Two Lines, No Visible Counter

```
for i in range(1, 6):
    print("Philip")
```

The `for` version is shorter, and arguably easier to read at a glance — there's less bookkeeping visible for a reader to have to verify. But "no visible counter" isn't the same as "no counter."

`range(1, 6)` produces a `range_iterator` object, and that object *is* keeping track of where it's up to internally — it just does it in code you never have to write or look at. The counter didn't disappear; it moved from your source code into the iterator's own implementation, written once (in C, as part of CPython itself) and reused by every `for` loop that ever runs.

⏱️ Which One Actually Runs Faster?

Rather than guess, here's an actual measurement — both loops doing the same job (summing the numbers from 0 up to 999,999), timed with Python's own `timeit` module, 20 runs each.

The Two Functions Being Compared

```
def while_loop():
    index = 0
    total = 0
    while index < 1_000_000:
        total += index
        index += 1
    return total

def for_loop():
    total = 0
    for i in range(1_000_000):
        total += i
    return total
```

✅ **VERIFIED RESULT (PYTHON 3.14, TIMEIT, 20 RUNS OF EACH)**

VERSION	TOTAL (20 RUNS)	PER RUN
`while_loop()`	0.7436s	37.18 ms
`for_loop()`	0.5321s	26.61 ms

The `for` loop came out about **1.4× faster** than the equivalent `while` loop for this task, on this machine. That's a real, measured number — not a rule that always holds exactly this ratio, but the direction (for loop faster) is consistent and explainable, not a fluke.

Why — Looking at the Actual Bytecode

Python doesn't run your source code directly; it first compiles it to bytecode, then executes that. Using the built-in `dis` module to disassemble both functions shows exactly where the extra work in the `while` version comes from.

while Loop's Per-Iteration Bytecode (condition check + increment, separately)

```
# the condition check, every iteration:
LOAD_FAST_BORROW    index
LOAD_SMALL_INT      5
COMPARE_OP          18 (bool(<))
POP_JUMP_IF_FALSE   (exit if false)

# ...loop body...

# the increment, every iteration:
LOAD_FAST_BORROW    index
LOAD_SMALL_INT      1
BINARY_OP           13 (+=)
STORE_FAST          index
JUMP_BACKWARD       (back to condition check)
```

for Loop's Per-Iteration Bytecode (one instruction does both jobs)

```
# the entire "get next value, or stop" step, every iteration:
FOR_ITER            (exit if exhausted)
STORE_FAST          i

# ...loop body...

JUMP_BACKWARD       (back to FOR_ITER)
```

The `while` loop needs four separate bytecode instructions just to check the condition, and another four just to increment — eight instructions of pure loop bookkeeping per pass, all executed one at a time by Python's interpreter loop. The `for` loop's `FOR_ITER` replaces *all of that*

with a single instruction, because it calls straight into the range iterator's own `__next__()` method — code written in C, not Python bytecode, and considerably faster per call. Fewer bytecode instructions dispatched per iteration is exactly why the measured numbers above come out the way they do.

> **⚠ DON'T OVER-GENERALISE THIS RESULT**
>
> This comparison is specific to CPython (the standard Python implementation) and to a loop body that does almost nothing besides the counting itself. In real code, the loop *body* usually does far more work than the loop mechanics — a database call, a network request, real computation — and that work dwarfs this difference completely. Rewriting a `while` loop as a `for` loop purely for a 1.4× speedup on the looping itself is rarely worth it on its own merits; write whichever one expresses the actual logic more clearly, and let that decide it.

🚫 The Loop Python Doesn't Have: do...while

Java and JavaScript both have a third loop form Python simply doesn't: `do { ... } while (condition);` — a loop that runs its body once, unconditionally, and only checks the condition afterward, at the bottom.

JAVA / JAVASCRIPT

`do { ... } while (cond);`

Body runs first, condition checked last. Guarantees at least one execution, no matter what the condition is.

```
do {
    console.log("Philip");
} while (x < 5);
```

PYTHON'S EQUIVALENT

`while True: ... if not cond: break`

There's no dedicated keyword — the idiom is a deliberately infinite loop with the condition check moved to wherever inside the body it needs to happen, ending in `break`.

```
while True:
    print("Philip")
    if not (x < 5):
        break
```

This omission is deliberate, not an oversight — it fits Python's general design philosophy of favouring one clear, explicit way of doing things over adding a new keyword for a case that can already be expressed cleanly with the constructs Python already has. A dedicated `do...while` keyword would only ever save a couple of lines over the `while True: ... break` pattern, and

Python's design has consistently leaned against adding syntax purely for that kind of convenience.

A Realistic Use: Input Validation That Must Run at Least Once

This is the single most common genuine "I want a do-while" moment in practice — prompting a user at least one time, and re-prompting only if what they typed doesn't pass a check.

```
while True:
    age = input("Enter your age: ")
    if age.isdigit():
        break
    print("Please enter digits only.")

print(f"Thanks, age recorded as {age}.")
```

So Are Python Programmers Actually Disadvantaged?

Not meaningfully, in practice. A genuine "must run the body at least once, check afterward" need does come up — the input-validation example above is a real, common one — but it's a relatively small slice of the loops most programs actually contain; most loops naturally check their condition *before* running, which is exactly what `while` already does directly. The `while True: ... break` pattern that fills the gap is well-established, instantly recognisable to any experienced Python developer, and arguably even a little more flexible than a real `do...while` — the exit check can be placed anywhere inside the body, not forced to sit at the very bottom. What you lose is a small amount of syntactic convenience, not any actual capability.

📋 Quick Reference — For vs While

QUESTION	ANSWER
Which is more concise for a known iteration count?	`for` — no manual init/increment to write
Which do you need for "loop until a condition changes"?	`while` — no fixed number of iterations, no collection to walk

QUESTION	ANSWER
Does `for` secretly still have a counter?	Yes — inside the iterator (e.g. `range` 's), just not written by you
Which runs faster in CPython for pure counting?	`for` — measured ~1.4× faster here; fewer bytecode instructions per iteration
Does that speed difference matter in real code?	Usually not — the loop *body*'s own work almost always dominates
Does Python have `do...while` ?	No — use `while True:` with a `break` where the check belongs

> **⚠ Gotchas for Java / JavaScript Programmers:**
>
> **Don't assume "fewer lines" always means "faster"** — *it happened to hold true in this lesson's own measurement, but conciseness and execution speed are genuinely separate properties. Judge each on its own terms.*
>
> **There's no labelled `break`** — *Java's `break outerLabel;` has no direct Python equivalent for exiting two nested loops at once. A flag variable, or wrapping the loops in a function and using `return`, is the idiomatic substitute.*
>
> **Reach for `while True: ... break` without hesitation** — *it can feel like a workaround if you're used to a real `do...while` keyword, but it's completely standard, idiomatic Python — not a hack.*