PYTHON LESSON

Boolean Logic Truthy Falsy

🔀 Boolean Logic — What Python Actually Considers True or False

Every value in Python has an opinion about whether it's "truthy" or "falsy" — not just actual `True`/`False` values. This lesson works through exactly what counts as which, using the prime-checker you wrote in class as the running example — including a direct answer to the question you asked about `not prime % i` vs `prime % i == 0`.

⚡ COMING FROM JAVA / JAVASCRIPT

Java has no truthiness at all. An `if` statement in Java requires an actual `boolean` — `if (x)` where `x` is an `int` simply won't compile. **JavaScript does have truthy/falsy**, much like Python — but the two languages don't agree on the details. JS's falsy list is `false, 0, -0, 0n, "", null, undefined, NaN`. Python's is different in a few important ways covered below, most notably: **empty containers** (`[]`, `{}`, `()`) are falsy in Python but truthy in JavaScript.

🚀 FASTAPI RELEVANCE

This comes up constantly in real endpoint code: `if not user:` after a database lookup that returns `None` when nothing's found, or `if not request.query_params.get("filter"):` to check whether an optional query parameter was actually supplied. Understanding exactly which values Python treats as "nothing here" is what makes those checks reliable instead of accidental.

📉 What Counts as Falsy

Only a specific, fixed set of values are falsy in Python. Everything else — genuinely everything else — is truthy.

THE NUMBER ZERO

`0, 0.0, 0j`

Every numeric type's own zero is falsy — integer, float, and even complex.

```
bool(0)     # False
bool(0.0)   # False
bool(-5)    # True -- negative numbers are still non-zero!
```

EMPTY CONTAINERS

`"", [], (), {}, set()`

An empty string, list, tuple, dict, or set is falsy. The SAME container with anything in it — even one falsy item — is truthy.

```
bool([])      # False
bool([0])     # True -- the list itself is non-empty
```

THE ABSENCE OF A VALUE

`None`

Python's own "nothing here" — what a function returns by default, and what a failed dictionary `.get()` returns.

```
user = db.find_user(id)
if not user:
    print("No user found")
```

THE LITERAL BOOLEAN

`False`

The obvious one — but worth including for completeness, since it's genuinely the same category as all the others above, not a special case.

```
bool(False)   # False
```

That's the complete list. There's no separate rule for "small" numbers, "short" strings, or anything else — a string containing the single character `"0"` is truthy (it's a non-empty string), and `-1` is truthy (it's non-zero). Truthiness in Python is about being empty/zero/absent, not about being small or negative.

❓ Your Question: `not prime % i` vs. `prime % i == 0`

Here's your own prime-checker's inner loop — this is exactly the truthy/falsy idea in action.

Your Original Code

```
prime = int(input("Enter a number to check if it is prime!"))
isprime = True

if prime in range(0,2):
  print ("0 and 1 are not considered to be prime!")
else:
  for i in range (3,prime):
    if not prime % i:
      isprime = False
```

✅ VERIFIED — THE TWO VERSIONS BEHAVE IDENTICALLY

`prime % i` is either `0` (i divides evenly — falsy) or some non-zero remainder (truthy). So `not prime % i` reads as "is the remainder falsy?" — exactly the same question as `prime % i == 0` asks directly. Run against every number from 2 to 199, both versions of your function classify every single one identically. There is **no behavioural difference** for this code.

So why might you choose one over the other?

- `prime % i == 0` **is more explicit.** Anyone reading it — including a future you — immediately knows what's being checked: "does the remainder equal zero." It doesn't require knowing Python's own truthy/falsy rules to understand at a glance.
- `not prime % i` **is more "Pythonic"** in the sense that using truthiness directly is a common, idiomatic style in Python (checking `if not my_list:` instead of `if len(my_list) == 0:` is genuinely preferred style) — but for a numeric remainder specifically, most style guidance leans toward the explicit `== 0`, since "is this number falsy" is a slightly unusual way to think about a mathematical remainder compared to "is this container empty."

⚠ THE ONE PLACE THIS WOULD HAVE MATTERED: OPERATOR PRECEDENCE

In Python, `%` binds *tighter* than `not` — so `not prime % i` is parsed as `not (prime % i)`, which is exactly what you want. Verified directly: `not 4 % 2` and `not (4 % 2)` both evaluate to `True`, while the deliberately different grouping `(not 4) % 2` evaluates to `0` — a completely different value. **This is genuinely important if you're coming from Java or JavaScript**, where `!` (the equivalent of `not`) binds extremely *tightly* — tighter than arithmetic operators. Literal JS syntax `!prime % i` would actually parse as `(!prime) % i`: convert `prime` to a boolean, negate it, then take THAT modulo `i` — nothing to do with checking whether `i` divides `prime` at all. Python's own precedence rules happen to make your code work as intended; the equivalent-looking JS wouldn't.

🔢 The Big Surprise: `bool` Is a Subtype of `int`

This has no equivalent in Java, and only a rough one in JavaScript — and it explains some code you'll see that looks strange at first.

Verified Directly

```
isinstance(True, int)              # True -- bool IS a kind of int
True == 1                      # True
False == 0                     # True
True + True                    # 2 -- booleans support real arithmetic
sum([True, True, False, True])  # 3 -- a real, common trick for counting Trues
```

That last line — `sum()` of a list of booleans — is a genuinely common, idiomatic Python pattern: counting how many items in a list satisfy some condition, by summing a list of `True`/`False` values directly, with no explicit counter variable at all.

📋 Quick Reference — Falsy Values in Python

VALUE	FALSY?	NOTE
`False`	✅ Falsy	The literal boolean
`0`, `0.0`, `0j`	✅ Falsy	Every numeric zero, across every numeric type
`""`	✅ Falsy	An empty string specifically — `" "` (a space) is truthy
`[]`, `()`, `{}`, `set()`	✅ Falsy	Empty containers only — one item, even a falsy one, makes them truthy
`None`	✅ Falsy	Python's own "nothing here" value
`-1`, `-0.5`	❌ Truthy	*Negative numbers are non-zero — a very common beginner assumption to double-check*
`"0"`, `"False"`	❌ Truthy	*Non-empty strings, regardless of what text they contain*
`[0]`	❌ Truthy	*The list has one item — its own value being falsy doesn't matter*

> **⚠ Gotchas for Java / JavaScript Programmers:**
>
> **Never write `if x == True:`** — *for a genuine boolean `x` this works, but if `x` could be any other truthy value (like `5`), `5 == True` is actually `False`, since `True` only equals `1` exactly. Just write `if x:`.*
>
> **An empty container is falsy, even if that surprises you coming from JS** — *`if []:` and `if {}:` both take the `else` branch in Python; in JavaScript, both would be truthy.*
>
> **Watch operator precedence with `not` and arithmetic** — *exactly the gotcha from your own code above: Python's `not` binds looser than `%`, `+`, and friends, but a literal port to JS's `!` would bind much tighter and silently mean something completely different.*