03

The Core Language

Values, types & control flow — for a JS developer

Chapter 3 · close to JavaScript, but the traps are different

03 The journey — five building blocks

Python's pieces look a lot like JavaScript's — but the rules underneath differ in ways that bite. Here are the five we'll walk through, one at a time.

By the end you can read the parsing, prompt-building and response-guarding that fills every agent codebase.

03 Names, not variables

There's no let, const or var — you just assign

A name in Python is just a LABEL pointing at a value. Assigning never makes a copy — it just points the label at the value. (This is the same as references in JS, but applied to everything.)

```
model = 'claude-opus-4'     # a label pointing at some text
model = 'claude-haiku'      # re-point the same label — totally fine
```

```
a = [1, 2, 3]
b = a           # b points at the SAME list, not a copy
b.append(4)
print(a)        # => [1, 2, 3, 4]   ← a changed too!
```

Watch out

Because b and a point at the SAME list, changing one changes the 'other'. There is no copy unless you ask for one.

03

Everything is an object

And a few assignment tricks you'll see constantly

Everything is an object

Whole numbers, text, functions, even 'nothing' (None) — every value has a type and its own built-in abilities. You can even store functions in a dictionary — that's how agents pick which tool to run.

```
x, y = 10, 20
x, y = y, x            # swap — no temp
needed
role, content = ('user', 'Hi')   # unpack a
pair
first, *rest = [1, 2, 3, 4]
# first = 1,  rest = [2, 3, 4]
```

This 'unpacking' — pulling several values out of a pair or list in one line — is everywhere in real Python. Get comfortable reading it.

03

The basic types

The five you'll meet on every line

```
int   —  whole numbers        e.g.  42,  10**100        never overflow (any size!)
float —  decimals             e.g.  0.7,  1e-4          same as JS numbers
bool  —  True / False         Capitalised.  Secretly a kind of int (True + True == 2)
str   —  text                 e.g.  'hi'                cannot be changed once made
None  —  'nothing'            the one and only 'no value'  (Python's only null)
```

Notice bool is a kind of int — so sum([True, False, True]) counts the True's. A real trick for counting matches.

03

None is the only 'nothing'

No 'undefined' to trip over

JavaScript has TWO empties — null and undefined — and endless confusion about which you got. Python has just ONE: None.

```
def maybe_tool():
    pass                # no 'return' line

print(maybe_tool())     # => None
```

A missing dictionary key, an unset value, a function with no 'return' — all of them give you None. One empty value to check for, not two.

03

Numbers — two traps for JS devs

Division, and gluing text to numbers

```
7 / 2      # => 3.5     '/' is ALWAYS decimal division
7 // 2     # => 3       '//' rounds DOWN to a whole number
-7 // 2    # => -4      (rounds toward the smaller number, not -3)
7 % 3      # => 1       remainder
2 ** 10    # => 1024    'to the power of'
```

Trap 1: in JS, 7/2 is 3.5 and you reach for Math.floor. In Python '/' is ALWAYS decimal, and '//' is the round-down one. Keep them straight.

Trap 2: Python will NOT glue text and numbers together.

'tokens: ' + 128 → ERROR
'tokens: ' + str(128) → ok
f'tokens: {128}' → ok (best)

JS quietly joins them; Python stops you — catching a real bug.

03 Text can't be changed in place

Every 'change' makes a NEW piece of text

```
name = 'claude'
name.upper()    # => 'CLAUDE'   (a NEW string)
print(name)     # => 'claude'   (the original is untouched)
```

So a line like `s.replace('a', 'b')` on its own does NOTHING useful unless you catch the result: `s = s.replace('a', 'b')`. This is a classic mistake to spot when reading code.

```
SYSTEM_PROMPT = '''You are a careful assistant.
Answer only from the provided context.
If unsure, say I do not know.'''    # triple quotes span lines
```

03 f-strings — the one to really know

The everywhere way to build text from values

```
model, temp, cost = 'opus', 0.7, 1234.5

f'Using {model} at temp={temp}'    # => 'Using opus at temp=0.7'
f'{cost:,.2f}'                     # => '1,234.50'  (commas + 2 decimals)
f'{0.8734:.1%}'                    # => '87.3%'     (as a percentage)
f'{temp=}'                         # => 'temp=0.7'  (prints name AND value)
```

Put an f before the quotes, then drop values (or any little calculation) inside {curly braces}. The bits after ':' control decimals, commas and percentages.

The {temp=} form prints both the name and its value — brilliant for logging and debugging.

03 Truthiness — 'empty' means false

The trap that catches every JS developer

```
These all count as FALSE ('falsy'):

False   None   0   0.0   ''   []   {}   ()

Everything else counts as true.
```

In JavaScript an empty list [] or object {} is TRUE. In Python they are FALSE — the exact opposite. This is the single most common bug when a JS dev writes Python.

```
if not response:          # catches '', None, and [] all at once
    raise ValueError('empty model response')

temperature or 0.7        # => 0.7 if temperature is missing... but ALSO
                          #    if it's a real 0.0 !  Use the safe form below:
temperature if temperature is not None else 0.7
```

03 'is' vs '==' — a subtle, important gap

Same value, or literally the same object?

```
==  asks:  same VALUE?

a = [1, 2]
b = [1, 2]
a == b    # => True   (same contents)
```

```
is  asks:  literally the SAME object?

a is b    # => False
(two different lists, even if equal)
```

The rule: use 'is' ONLY for None → if config is None: Everywhere else, use '==='. Writing 'is' to compare numbers or text (count is 0, name is 'user') works by luck for small values and fails silently for bigger ones — a real bug. Flag it on sight.

03

Control flow — and the big loop change

Blocks are made by indenting, not by braces

```
if score > 0.9:
    tier = 'high'
elif score > 0.5:          # 'else if' is spelled elif
    tier = 'medium'
else:
    tier = 'low'

label = 'pass' if score >= 0.5 else 'fail'   # no ' ? : ' — reads left to right
```

```
for msg in messages:              # you loop over the VALUES,
not indexes
    print(msg)
for i, msg in enumerate(messages):   # want the index too?
use enumerate
    ...
for a, b in zip(names, scores):      # walk two lists
together
```

The big change from JS: a 'for' loop hands you each VALUE directly, not a counter. No 'for (let i...)'. You always loop over something.

03 match / case — a smarter switch

Modern Python, used a lot for message/tool handling

```
def handle(event):
    match event:
        case {'type': 'text', 'text': body}:      # matches AND grabs 'body'
            return body
        case {'type': 'tool_use', 'name': name}:
            return f'calling {name}'
        case {'type': 'error', 'code': code} if code >= 500:   # extra 'if' guard
            return 'server error, retry'
        case _:                                   # _ = anything else
            return 'unknown'
```

It's like a switch, but it can look INSIDE the data, pull out the pieces you name, and even add an 'if' condition. 'case _' is the catch-all at the end.

03 The JS → Python cheat sheet

`&&  ||  !`	→	`and  or  not`
`===`	→	`==` (no coercion; no `===` needed)
`c ? a : b`	→	`a if c else b`
`x == null`	→	`x is None`
`Math.floor(a/b)`	→	`a // b`
`for..of  /  forEach`	→	`for x in seq  /  enumerate(seq)`
`switch`	→	`match / case`

Bugs to hunt when reading:

1. Looping by index when a plain 'for x in items' would do.
2. A string .replace(...) on its own line — does nothing (text can't change in place).
3. 'is' used to compare numbers or text — a latent bug.
4. 'value or default' quietly swallowing a real 0 or ''.

The teaching — Python's atoms look like JavaScript's, but the rules underneath differ. Learn the handful of gaps here and you can read — and judge — the code the whole AI field is written in.