05

Functions & Decorators

Arguments, scope, closures — and the one truly alien thing: decorators

Chapter 5 · learn to read these and half of an agent codebase stops looking like magic

05 The journey — five moving parts

Functions look familiar coming from JavaScript, but the calling rules are richer and there is one alien construct at the end. Here are the five stops, one at a time.

Every LLM SDK, every agent framework and every FastAPI service leans on the features in this chapter.

05

def, return, and docstrings

No braces — the indented block IS the body

'def' defines a function; 'return' hands back a value. End with no return and it hands back None — Python's one and only 'nothing'. There is no 'undefined'.

```
def greet(name):
    return f'Hello, {name}'     # f-string, from chapter
3

def log(prompt):
    print(prompt)               # no return line...
log('hi')                       # => prints, then returns
None
```

The docstring

The first line can be a triple-quoted string — the docstring. It is NOT a comment: it lives on the function as name.__doc__. Agent frameworks READ it (plus your type hints) to build the tool schema the model sees. A vague docstring makes the model worse. Treat it as prompt engineering.

JS to Python: 'def name(args):' replaces 'function name(args) {}'. No 'function' keyword, no hoisting — a function must be defined ABOVE where it runs. JSDoc is optional tooling; a docstring is a real runtime attribute frameworks introspect.

05

Arguments: by position or by name

Passing by name is idiomatic and everywhere

```
def chat(model, prompt, temperature=0.7):
    ...

chat('gpt-4o', 'hello')                    # positional
chat(prompt='hello', model='gpt-4o', temperature=0.2)   # by name (keyword)

client.chat.completions.create(model=..., messages=..., temperature=...)
```

Why by name (keyword arguments)?

Order stops mattering, intent is explicit, and calls survive reordering. Real SDK calls almost always spell out model=, messages=, temperature=, max_tokens=. Defaults (temperature=0.7) work just like JS defaults.

JS to Python

JavaScript fakes this with an options object: chat({ model, prompt, temperature }). Python has it as real syntax — temperature=0.2 is a language feature, not a destructured object.

05

The mutable default trap

The single most-flagged Python bug in code review

A default value is created ONCE, when the function is defined — not on each call. So a list, dict or set default is SHARED across every call.

```
def add_message(msg, history=[]):    # BUG
    history.append(msg)
    return history

add_message('a')    # => ['a']
add_message('b')    # => ['a', 'b']  <- 'a'
leaked across calls!
```

```
def add_message(msg, history=None):
    if history is None:
        history = []        # fresh list
every call
    history.append(msg)
    return history      # => ['a'] then ['b']
fixed
```

Reading & judging — any default that is not an immutable primitive (None, a number, a string, a tuple) is suspect. def f(x, items=[]), f(x, cfg={}), f(x, seen=set()) are all bugs unless a shared cache was truly intended — which it almost never is. In an agent this looks like conversation history bleeding between unrelated requests. A legitimate reason to send a PR back.

05

*args and **kwargs

Collect extra arguments — and spread them back out

```
def call_llm(prompt, *args, **kwargs):
    print(args)      # extra POSITIONALS as a tuple
    print(kwargs)    # extra KEYWORDS as a dict
    return kwargs.get('temperature', 0.7)

call_llm('hi', 'x', 'y', temperature=0.2, top_p=0.9)
# args   => ('x', 'y')
# kwargs => {'temperature': 0.2, 'top_p': 0.9}
```

```
def make_request(model, **opts):
    return {'model': model, **opts}

defaults = {'temperature': 0.7,
'max_tokens': 512}
make_request('gpt-4o', **defaults)  #
spread a dict IN
```

The everywhere pattern: accept **kwargs and forward them straight to the underlying client — so new provider parameters work without editing your wrapper. * collects a tuple, ** collects a dict; the same stars SPREAD on the way out.

JS to Python

*args is like rest params (...args). f(*mylist) is like f(...myArray). But **kwargs has no JS twin — rest params for NAMED arguments, and f(**mydict) spreads a dict into keywords, which JS cannot do.

05

The / and * in a signature

Two markers that pin HOW an argument may be passed

A bare '/' means: everything before me must be passed by POSITION. A bare '*' means: everything after me must be passed by NAME (keyword). Well-designed SDKs use both.

```
def embed(text, /, *, model, normalize=True):
    ...

embed('hello', model='text-embedding-3-small')   # ok
embed(text='hello', model=...)   # TypeError: text is positional-only
embed('hello', 'text-embedding-3-small')   # TypeError: model is keyword-only
```

Why bother? 'text' before / can never be named, so the author can rename it later without breaking callers. 'model' and 'normalize' after * can never be positional, so calls stay readable and the author can reorder them safely.

05

Type-annotated signatures (a preview)

Read them as documentation the tooling can act on

```
def chat(prompt: str, temperature: float = 0.7) -> str:
    return f'...{prompt}...'
```

What the annotations mean

': str', ': float' and '-> str' are TYPE HINTS. Python does NOT enforce them at runtime — pass a wrong type and it still runs. They exist for humans, editors and type checkers (mypy, pyright).

Why you care

Frameworks generate tool schemas from these hints. Together with the docstring, they are what an agent framework turns into the JSON the LLM sees. Good hints = better tool calls. (Full type system comes later.)

05

Scope — the LEGB rule

How Python decides which name you mean

To resolve a name, Python walks outward: Local, then Enclosing, then Global, then Built-in (LEGB). It stops at the first match.

```
MODEL = 'gpt-4o'            # Global

def outer():
    provider = 'openai'     # Enclosing (for inner)
    def inner():
        temp = 0.5          # Local
        return f'{provider}/{MODEL} temp={temp}'
    return inner()          # => openai/gpt-4o temp=0.5
```

Assigning to a name inside a function makes it LOCAL by default. To rebind an OUTER name you must declare intent: 'global' for module level, 'nonlocal' for an enclosing function. Without them, counter += 1 raises UnboundLocalError. Avoid 'global' in real code — it is a smell, same as leaning on JS globals.

JS to Python: LEGB is the JS scope chain, and Python closures behave like JS closures. Difference — JS declares with let/const/var; Python assignment implicitly declares local, so you say 'global'/'nonlocal' to reach outward and rebind.

05

The late-binding closure gotcha

Closures capture the VARIABLE, not its value at the time

```
funcs = [lambda: i for i in range(3)]
[f() for f in funcs]    # => [2, 2, 2]
# all three lambdas close over the SAME i, which ends at 2
```

```
funcs = [lambda i=i: i for i in range(3)]   # bind per-iteration
[f() for f in funcs]    # => [0, 1, 2]  fixed
# the default i=i is evaluated at DEFINITION time, freezing each value
```

Reading & judging — this is the Python twin of the classic 'for (var i...) setTimeout' puzzle. In async or retry code you will see loops that build handlers or tasks; if each should capture a different value but there is no 'x=x' default (or no factory function), suspect this bug.

05 Lambdas — one-expression functions

Their real home is a key= argument

```
square = lambda x: x * x      # rarely worth naming
square(5)                     # => 25
```

```
runs = [{'model': 'gpt-4o', 'cost': 0.03},
        {'model': 'haiku',  'cost': 0.001},
        {'model': 'opus',   'cost': 0.09}]

cheapest = min(runs, key=lambda r: r['cost'])   # =>
haiku
by_cost  = sorted(runs, key=lambda r: r['cost'])
```

A lambda is expression-only — no return, no statements, just a value. It is like the JS arrow x => x*x, but with no braces and no multi-line body.

Reading & judging — the moment you want a line of logic, write a 'def'. A lambda spanning half a screen is a smell: it can not have a docstring, can not be tested by name, and hurts readability. Pull it out into a named function.

05

Functions are values

Pass them, return them, store them — the base of decorators

```
def retry_call(fn, prompt):          # a function TAKES a
function
    for _ in range(3):
        try:
            return fn(prompt)
        except Exception:
            continue

retry_call(str.upper, 'hi')          # => HI
```

```
list(map(lambda x: x*2, nums))      #
functional style
[x * 2 for x in nums]               #
idiomatic Python

list(filter(lambda x: x%2==0, nums))
[x for x in nums if x % 2 == 0]
```

Reading & judging — 'map'/'filter' with a lambda usually should have been a comprehension. Seeing list(map(lambda ...)) is a mild sign the author is writing JavaScript in Python. 'reduce' in particular is discouraged where a plain loop or sum/math.prod reads more clearly.

05

Decorators from first principles

A function that takes a function and returns a wrapped one

```
def shout(fn):
    def wrapper(*args, **kwargs):
        return fn(*args, **kwargs).upper()
    return wrapper

@shout                  # pure sugar for:  greet =
shout(greet)
def greet(name):
    return f'hi {name}'

greet('ada')            # => HI ADA
```

That is the ENTIRE idea. The @shout line just reassigns greet to the wrapped version. The *args, **kwargs in the wrapper is what lets ONE decorator wrap ANY function, whatever its signature.

functools.wraps — always use it

Without it the wrapper REPLACES your function's identity: greet.__name__ becomes 'wrapper' and the docstring vanishes — breaking the exact introspection agent frameworks rely on. A hand-written decorator missing @functools.wraps is a real (if quiet) bug. Flag it.

JS to Python: vanilla JavaScript has no decorators. The closest analog is a higher-order wrapper — const greet = shout(originalGreet) — which is literally what @shout desugars to. TypeScript's @decorator and Angular/NestJS routing are the direct cousins.

05

Decorators with arguments

To write @retry(times=3) you need one more layer

```
import functools, time

def retry(times=3, delay=0.5):
    def decorator(fn):
        @functools.wraps(fn)
        def wrapper(*args, **kwargs):
            for attempt in range(times):
                try:
                    return fn(*args, **kwargs)
                except Exception as e:
                    time.sleep(delay)
            raise e
        return wrapper
    return decorator

@retry(times=3, delay=0.2)
def call_model(prompt): ...   # flaky, rate-limited API
```

Three nested layers

retry(...) returns 'decorator', which receives call_model and returns 'wrapper'. The outer layer exists purely to capture the arguments (times, delay). Retry decorators are everywhere in LLM code because provider APIs rate-limit and time out constantly.

@timing is the other canonical one — start a timer, run the function in a try/finally, print how long it took. Same shape; it slots in with one line above a def.

05 The decorators you must recognize

```
@property              computed attribute — obj.tokens LOOKS like a
field, runs a method

@staticmethod / @classmethod   no 'self' / takes 'cls' — alternate
constructors like from_config(...)

@lru_cache / @cache    memoize by arguments — cache embeddings,
tokenizer loads, config reads

@dataclass             auto-writes __init__, __repr__, __eq__ (the
classes chapter)

@app.get / @app.post   FastAPI — registers the function as an HTTP
route; the decorator IS the router

@tool / @agent         LangChain, LlamaIndex — registers an LLM-
callable tool from docstring + hints

@pytest.fixture / @mark    test setup and test tagging
```

Reading & judging

Decorators HIDE control flow — @retry silently loops, @cache silently skips, @app.post means the framework calls it, never you. When a function 'is not being called but clearly runs', look UP at its decorators.

Watch for: over-clever stacks (order matters, unclear), a missing @functools.wraps, and side effects at IMPORT time — a factory doing real work (network, file reads) when the module loads, turning a simple import into a landmine.

The teaching — a decorator is just a function that wraps a function. Learn to read that one pattern and the retry, timing, routing and tool-registration magic across every agent codebase turns into plain Python you can judge.