06

Classes & Dataclasses

Objects the Pythonic way — for a JS developer

Chapter 6 · self, magic methods and @dataclass make AI code readable

06 The journey — five things to master

You already know classes from JavaScript. Python's are close in spirit but differ in a few ways that show up on almost every line of AI code. Here are the five we'll walk through, one at a time.

Master these and most GenAI codebases — agents, tools, configs, messages — become readable at a glance.

06

A class and its constructor

__init__ runs when you build the object

A class is a template for making objects. The __init__ method (say 'dunder init') is the constructor — it runs when you call ChatSession(...) and fills in the new object. It does NOT return anything; Python hands you the object as self.

```
class ChatSession:
    def __init__(self, model, system='You are helpful.'):
        self.model = model              # instance attribute
        self.system = system
        self.messages = []              # each session gets its OWN list

    def add(self, role, content):       # methods take self, explicitly
        self.messages.append({'role': role, 'content': content})

session = ChatSession('claude-opus-4')
session.add('user', 'Hello')
```

06

Why write self at all?

JS gets this for free — Python makes it a plain parameter

One simple rule

session.add('user', 'Hi')

is just shorthand for

ChatSession.add(session, 'user', 'Hi')

The object is passed as the FIRST argument, like any other. So you always WRITE self in the definition — but you never pass it yourself.

JS to Python

JS 'this' is bound by the call site — easy to lose inside a callback. Python's self is an ordinary parameter that can never be the wrong object. No this-rebinding foot-gun.

Also: there is NO 'new' keyword. You call the class directly — Foo(3) — not new Foo(3).

Nothing is implicit or bound by magic — which is exactly why Python classes are easy to reason about.

06

Instance vs class attributes

A shared list is one of the most common Python bugs

An attribute set with self.x inside __init__ belongs to THAT object. An attribute set in the class body is shared by EVERY instance — one single copy.

```
class Agent:
    provider = 'anthropic'    # class attribute — one copy,
shared (fine)
    tools = []                # DANGER: also shared across
all instances!

    def __init__(self, name):
        self.name = name      # instance attribute — one per
object

a, b = Agent('planner'), Agent('coder')
a.tools.append('search')
print(b.tools)    # => ['search']   b sees a's change — same
ONE list
```

Reading & judging

A mutable class attribute (= [] or = {}) that instances later .append or index-assign is a classic accidental-sharing bug.

Fix: make a fresh list per object inside __init__ (self.tools = []). Constants — strings, numbers, tuples — are safe because they can't be changed.

06

Magic methods — objects that plug into syntax

'Dunder' = double-underscore, __like_this__

These specially-named methods let YOUR object hook into Python's built-in syntax — printing, ==, len(), indexing, looping. This is where Python's OOP earns its keep in AI libraries.

```
class Prompt:
    def __repr__(self):    # for developers: REPL, logs, debuggers
        return f'Prompt(text={self.text!r}, tokens={self.tokens})'
    def __str__(self):     # for users: print() and f-strings
        return self.text
    def __eq__(self, other):   # defines what == means for Prompts
        return self.tokens == other.tokens
    def __len__(self):  return len(self.tokens)      # enables len(p)
    def __getitem__(self, i): return self.tokens[i]  # enables p[0]
    def __iter__(self): return iter(self.tokens)     # enables for x in p
```

06

__repr__ vs __str__

Two ways to turn an object into text

__repr__ — the debugging face

Unambiguous, for developers. Shows up in the REPL, logs, stack traces, and inside lists. Aim to make it look like the code that would rebuild the object.

If you define only one, define this one — Python falls back to it when __str__ is missing.

__str__ — the friendly face

Human-readable, for print(), f-strings and end users. The pretty output a person should see.

The !r inside {self.text!r} inserts the repr of a value (with quotes) — so the debug output is copy-pasteable.

Reading & judging — an object with no __repr__ prints as <myapp.Agent object at 0x10f3c2a50>: undebuggable. Hex-address blobs in your logs mean a missing __repr__ — a real code-quality smell in anything you'll have to operate.

06

Two dunders all over agent code

Callable objects, and the 'with' statement

```
class Tool:
    def __init__(self, name, fn):
        self.name = name; self.fn = fn
    def __call__(self, **kwargs):     # makes the
OBJECT callable
        return self.fn(**kwargs)

search = Tool('search', run_search)
search(query='python')    # used like a
function...
search.name               # ...but still an object
with state
```

```
class Timer:
    def __enter__(self):          # runs at
'with' entry
        print('start'); return self
    def __exit__(self, *a):       # runs at
exit, even on error
        print('done')

with Timer():
    print('working')
# => start / working / done
```

__call__ is why an agent's tool can be USED like tool(...) while still BEING an object with a .name, a schema and state. __enter__ / __exit__ power the 'with' statement — the standard way to manage resources like open clients or streaming responses (full detail in Chapter 7). Operator overloading is the same idea: __add__ defines +, __lt__ defines <. Libraries use it sparingly — only when the operator's meaning is obvious.

06

property, classmethod, staticmethod

Three decorators you'll read constantly

```
class Usage:
    def __init__(self, prompt_tokens, completion_tokens):
        self.prompt_tokens = prompt_tokens
        self.completion_tokens = completion_tokens

    @property
    def total_tokens(self):              # a COMPUTED attribute
        return self.prompt_tokens + self.completion_tokens

    @classmethod
    def from_response(cls, resp):        # alternative
constructor
        return cls(resp['input_tokens'], resp['output_tokens'])

    @staticmethod
    def price(tokens, per_million):      # plain function,
namespaced
        return tokens / 1_000_000 * per_million
```

@property
Read like an attribute — u.total_tokens with NO parentheses — but recomputed each time. Great for derived values.

@classmethod
The idiomatic 'named constructor' — cls is the class itself. You'll see Model.from_config(...), Client.from_env() everywhere, because Python allows only one __init__.

JS to Python
@property = get/set. @classmethod ~ a static from(...) factory, but cls makes it subclass-aware. @staticmethod = JS static method.

06 Inheritance, super() — and why composition wins

A subclass extends a parent; super() calls back to it

```
class Agent:
    def __init__(self, name):
        self.name = name
    def act(self):
        return f'{self.name} thinking'

class ToolAgent(Agent):              # ToolAgent IS-A Agent
    def __init__(self, name, tools):
        super().__init__(name)       # run the parent __init__
FIRST
        self.tools = tools
    def act(self):                   # override the parent
method
        return super().act() + f' with {len(self.tools)}
tools'

isinstance(ToolAgent('x', []), Agent)    # => True
```

Python allows multiple parents; when an attribute could come from several, it's resolved by the MRO (Method Resolution Order), a fixed ordering you can inspect with ClassName.__mro__.

Composition often wins

Instead of inheriting from several parents, an object HOLDS its collaborators:

```
class RagAgent:
    def __init__(self, retriever,
model):
        self.retriever =
retriever
        self.model = model
```

Easier to test, swap and reason

Reading & judging — inheritance more than 2–3 levels deep, or a subclass that overrides most of its parent, usually means inheritance where composition fits. A missing super().__init__ leaves objects half-built. Prefer isinstance(x, Agent) over type(x) == Agent — it respects subclasses.

06

@dataclass — the modern default for data holders

It writes __init__, __repr__ and __eq__ for you

Most classes in AI code just bundle data. Writing the constructor, the repr and the equals by hand is pure boilerplate. @dataclass generates all three from your type-annotated fields.

```
from dataclasses import dataclass, field

@dataclass
class Message:
    role: str
    content: str
    name: str | None = None          # a field with a default

@dataclass(frozen=True)              # frozen => immutable, can't reassign
class ModelConfig:
    name: str
    temperature: float = 0.7
    stop: list[str] = field(default_factory=list)   # SAFE mutable default

Message('user', 'hi') == Message('user', 'hi')   # => True (free __eq__)
```

06

Dataclass: the two things to internalise

And where dataclasses stop and Pydantic begins

1. Safe empty defaults

field(default_factory=list) calls list() FRESH for each object — dodging the shared-mutable-default bug from earlier. Dataclasses even raise an error if you try stop: list = [] directly. A nice guardrail.

2. frozen=True

Makes instances immutable (can't be changed) and hashable — ideal for config objects and cache keys, and aligned with the 'new objects, never mutate' discipline these codebases favour.

Reading & judging — dataclasses annotate types but do NOT validate them: Message(role=123, content=None) builds happily. Use them for internal, TRUSTED data. When data crosses a trust boundary — API bodies, LLM tool-call arguments, config files — reach for Pydantic, which validates and coerces (Chapter 8). A hand-written class that's just __init__ assigning self.x = x plus a __repr__ should almost always be a @dataclass.

06

Protocols vs ABC — two kinds of interface

Duck typing: if it walks like a duck...

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class ChatModel(Protocol):
    def chat(self, prompt: str) -> str: ...

class AnthropicModel:           # does NOT inherit ChatModel
    def chat(self, prompt):
        return f'[claude] {prompt}'

# any object with a matching .chat() counts as a ChatModel
isinstance(AnthropicModel(), ChatModel)    # => True
```

Protocol = structural

Anything shaped right counts — no base class, no registration. Great for pluggable model / tool backends you don't own. This is the direct analogue of a TS interface.

ABC = nominal

abc.ABC is an explicit interface a class must inherit; its @abstractmethod raises a TypeError at instantiation if you forget to implement it. Like a TS abstract class you must extend.

Use a Protocol to accept anything shaped right; use an ABC when you're defining a base others must explicitly extend and want a hard error if they forget a method.

06 When NOT to reach for a class

Use a class (or dataclass) when...

- state is shared by several methods
- multiple interchangeable implementations sit behind one interface
- objects benefit from __repr__ / __eq__

Reach for something simpler when...

- a stateless transformation → a plain function
- a loose, short-lived bag of data → a plain dict
- a class with one method and no state → a function in a costume

Bugs to hunt when reading: a mutable class attribute (= [] / = {}) shared by accident · hex-address blobs in logs = missing __repr__ · a subclass with no super().__init__ · inheritance 3+ levels deep where composition fits · a @dataclass fed untrusted input (that's Pydantic's job).

The teaching — self written out, magic methods, and @dataclass are the three things that make Python OOP readable. Judge for the right altitude: enough structure to make state and interfaces explicit, never so much that a two-line transform hides inside three classes.