07

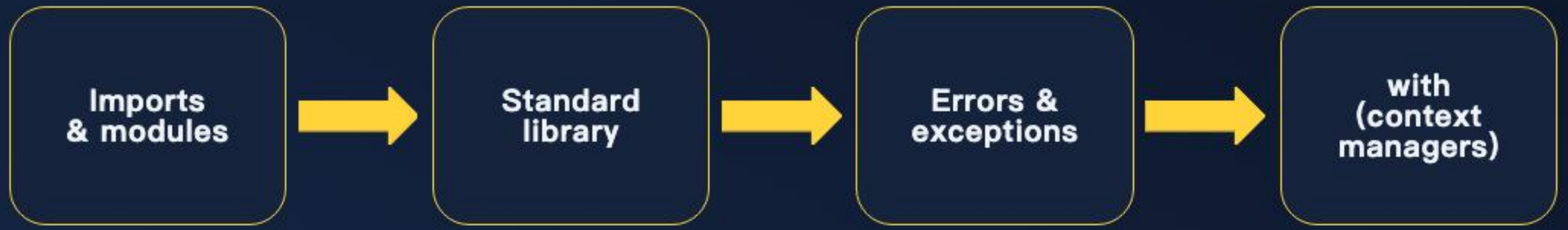
Modules, Errors & Context Managers

How files pull each other in, lean on batteries, and fail safely

Chapter 7 · learn to read the three shapes that fill every agent codebase

07 The journey — four recurring shapes

You can read a lot of GenAI code once you recognise how files pull each other in, how they lean on the built-in library, and how they fail safely. Four stops, one at a time.



By the end you can read the imports, the stdlib calls, the try/except guards and the with-blocks that make up real agent code.

07

A module is just a .py file

And importing it RUNS it — once

A module is one .py file. A package is a folder of them. When you write 'import agent', Python finds agent.py, runs it top to bottom ONCE, and hands you the resulting names. That 'runs it once' part surprises JS developers.

```
# config.py
print('loading config...')      # runs the
FIRST time it's imported
MODEL = 'claude-opus-4'
```

```
# main.py
import config   # => loading config...
(printed on import)
import config   # silent — already cached in
sys.modules
print(config.MODEL)    # => claude-opus-4
```

Everything at the top level runs on that first import — function and class definitions, but also any loose statements (that print). Import the same module twice and the second time is silent: it is cached.

07

The four import forms you'll meet

One of them you should avoid

```
import json                       # bind the module; use as json.loads(...)
from json import loads, dumps     # bind two names directly; use loads(...)
import numpy as np                # alias — the near-universal np / pd habit
from json import *                # bind EVERYTHING public — AVOID (see right)
```

JS -> Python

'import config' binds the whole module object (like import * as config). 'from config import MODEL' is like import { MODEL }. There is no default export — every top-level name is a named export.

Why avoid 'import *'

It dumps every public name into your file. Now a reader (and the linter) can't tell where a name like parse(...) came from, and it can quietly shadow a builtin. Prefer explicit imports.

07

The __name__ == '__main__' guard

Library code that also has a script mode

Every file has a magic variable called __name__. RUN the file directly and Python sets it to the text '__main__'. IMPORT the same file and it is set to the module's own name instead. So this block runs only on direct execution.

```
def summarize(text):
    return text[:100] + '...'

if __name__ == '__main__':
    # a quick self-test / CLI entry — skipped when imported
    print(summarize('some long transcript'))
```

Importing this module gives you 'summarize' WITHOUT running the demo. It is THE idiom for a file that is both a library and a runnable script. The JS cousin is require.main === module, or import.meta.main.

07

Reading & judging imports

Two smells worth spotting

Circular imports = a design smell

If a.py imports b at the top and b.py imports a at the top, whichever loads first hits a HALF-BUILT module — you get an ImportError or a mysterious None. It usually means the two files should be one, or need a shared third. Seeing an import moved INSIDE a function (a lazy import) is often a patch for exactly this.

Red flag: work at import time

'import config' should NOT open a socket, write a file, or read env vars on its own. If simply importing a module does real work, tests become slow and order-dependent. Prefer doing that work INSIDE functions the caller chooses to invoke. Also: never name your file json.py next to code that imports the stdlib json — the first match on the path wins, and you get shadowing bugs.

07

Standard library, part 1

You don't memorise these — you recognise them

```
from pathlib import Path        # modern file paths, prefer over os.path

system_file = Path('prompts') / 'system.txt'   # '/' joins segments
if system_file.exists():
    text = system_file.read_text(encoding='utf-8')  # whole file -> str
Path('out').mkdir(exist_ok=True)   # make dir, no error if it's there
```

```
import os
key = os.environ['OPENAI_API_KEY']   # raises KeyError if
MISSING
key = os.environ.get('OPENAI_API_KEY')  # returns None if
missing
model = os.environ.get('MODEL', 'claude-opus-4')   # with a
default
```

JS -> Python

os.environ['KEY'] is process.env.KEY — but the bracket form THROWS on a missing key (good at startup: a missing key is fatal). The .get form quietly returns None, like JS.

07

json — read tool-calls, write output

Constantly used with LLMs — so guard the parse

```
import json
data = json.loads('{"tool": "search"}')    # str -> dict
print(data['tool'])                 # => search
pretty = json.dumps(data, indent=2)    # dict -> readable str
```

```
raw = '{"score": 0.9, oops}'    # model produced BROKEN
JSON
try:
    result = json.loads(raw)
except json.JSONDecodeError as e:
    print(f'invalid JSON at position {e.pos}')
```

Red flag

LLMs are ASKED for JSON but sometimes emit malformed text. A bare json.loads(model_output) with no 'except json.JSONDecodeError' around it is a latent crash. Always guard the parse.

07 Standard library, part 2 — recognise the shape

```
collections   —  defaultdict, Counter, namedtuple   (auto-default keys, tallies, tiny
records)

itertools     —  chain, islice, groupby   (lazy tools over streams — no big lists built)

functools     —  cache, partial, wraps, reduce   (memoise, pre-fill an arg, fold to one
value)

datetime, time   —  timestamps, and time.sleep(n) — the pause you see between retries

enum.Enum     —  named constant sets   (class Role(Enum): USER = 'user')

re            —  regular expressions; the pattern lives in a raw string  r'...'
```

A writer who reaches for defaultdict / Counter instead of hand-rolling 'if key not in d' knows the stdlib. New code should prefer pathlib over os.path string paths.

07

Errors — try / except / else / finally

Python signals failure by RAISING; you CATCH it

```
try:
    n = int('not a number')
except ValueError as e:       # catch a SPECIFIC type
    print(f'bad input: {e}')
else:
    print('runs only if NO exception')    # optional
finally:
    print('always runs — cleanup')         # optional
```

The four parts

except SpecificError — handle one kind of failure.

else — runs only if try raised NOTHING.

finally — runs no matter what, even if the error keeps propagating. Cleanup goes here.

JS -> Python: this is try/catch/finally, plus an else clause JS lacks.

Catch a NARROW type (ValueError, KeyError, TimeoutError — all inherit from Exception) so you only intercept failures you actually understand.

07 Raise, chain, custom types, assert

```
raise ValueError('temp must be 0..2')   # raise your own

try:
    cfg = json.loads(raw)
except json.JSONDecodeError as e:
    raise RuntimeError('config is not JSON') from e   #
keep original

class RateLimitError(Exception):
    '''raised when the API returns HTTP 429'''
```

```
assert 0 <= temp <= 2       # fine — an internal sanity
check
if not api_key:             # correct way to validate REAL
input
    raise ValueError('API key required')
```

Chaining with 'from'

When you catch one error and raise a clearer one, 'from e' keeps the original in the traceback — invaluable for debugging. Custom exception classes let callers catch YOUR failures precisely.

assert is dev-only

Run Python with the -O (optimise) flag and it STRIPS every assert. So never use assert to validate untrusted input or enforce security. Use a real 'if ... raise' instead.

07

The shape you'll read constantly

Catch the specific things; let the unexpected propagate

```
def call_model(client, prompt, retries=3):
    for attempt in range(retries):
        try:
            resp = client.complete(prompt, timeout=30)
            return json.loads(resp.text)    # parse output
        except TimeoutError:
            time.sleep(2 ** attempt)    # backoff, retry
        except RateLimitError:
            time.sleep(5)               # wait, retry
        except json.JSONDecodeError as e:
            raise ModelOutputError('not JSON') from e
    raise TimeoutError('failed after retries')
```

The #1 red flag: the silent swallow

```
except Exception:
    pass
```

The failure vanishes with no log and no re-raise — a failed API call then looks identical to a success. A bare 'except:' is worse: it even swallows Ctrl-C and exit. Legit broad catches LOG and RE-RAISE.

07

Context managers — the 'with' statement

Setup and teardown, guaranteed as a pair

```
with open('transcript.txt', 'r', encoding='utf-8') as f:
    text = f.read()
# file is CLOSED here automatically — even if read() had raised
```

```
f = open('transcript.txt')
data = process(f.read())   # if this raises, file NEVER
closes -> leak
f.close()
```

Reading & judging: a manual open(...) or client = httpx.Client() with NO matching 'with' (and no .close() in a finally) is a resource-leak smell — handles and sockets pile up.

Without 'with', a leak is one early return or exception away. The 'with' version closes the file no matter how the block exits.

Multiple contexts stack in one line:
with open('in') as s, open('out','w') as d:

Same pattern for HTTP clients, locks, and DB sessions.

07

Writing your own context manager

Two ways: a class, or a single function

```
class Timer:
    def __enter__(self):
        self.start = time.perf_counter()
        return self          # bound after 'as'
    def __exit__(self, exc_type, exc_val, exc_tb):
        dur = time.perf_counter() - self.start
        print(f'took {dur:.3f}s')   # runs even if body
raised
        # return False (default) -> exception still
propagates
```

```
from contextlib import contextmanager

@contextmanager
def timer(label):
    start = time.perf_counter()
    try:
        yield            # with-body runs here
    finally:
        print(f'{label}: done')   # guaranteed
teardown
```

A class becomes a context manager with __enter__ (setup, returns the object bound after 'as') and __exit__ (teardown, runs on the way out — even on exception). The lighter way is @contextmanager: everything before 'yield' is setup; the 'finally' after it is guaranteed teardown. Returning True from __exit__ SUPPRESSES the exception — usually you don't, so it propagates. A custom __exit__ that returns True unconditionally is suspicious: it silently eats EVERY error in the block.

07 The JS -> Python cheat sheet

`require / import * as x`	->	`import x`
`import { a } from '...'`	->	`from mod import a`
`require.main === module`	->	`if __name__ == '__main__':`
`try / catch / finally`	->	`try / except / else / finally`
`throw err`	->	`raise Err(...)`
`process.env.KEY`	->	`os.environ['KEY']` (throws if missing)
`using / try..finally`	->	`with ...:` (context manager)

Bugs to hunt when reading:

1. 'import *' — you can't trace where a name came from.
2. Real work at import time (network, file, env) — slow, order-dependent tests.
3. except Exception: pass — the silent swallow.
4. open(...) or a client with no 'with' — a leak.
5. assert used to check untrusted input — stripped by -O.

The teaching — Three shapes recur in every agent codebase: how files pull each other in (imports), how they lean on the standard library, and how they fail safely (exceptions and with). Learn to read these and most GenAI code becomes legible.