PYTHON FROM FOUNDATIONS TO PRODUCTION

A practical learning journey built around one evolving workplace story

Written by the Team of Swatantra

SWATANTRA.ai

Preface

Python is one of the easiest programming languages to begin using, but learning enough syntax to make a program run is not the same as learning how to build software that can be understood, tested, changed, and operated with confidence. This book was created to bridge that distance without forcing a beginner to cross it in a single leap.

The progression is deliberate. The early chapters begin with ordinary programs, values, decisions, functions, collections, files, and errors. Only after those foundations become familiar does the book move into object design, reliability, concurrency, APIs, packaging, deployment, and security. The aim is not to make Python appear simple by hiding its deeper ideas, nor to make it appear difficult by introducing advanced mechanisms before they have a purpose. Each topic arrives when the problems introduced earlier make the next idea useful.

A single evolving workplace story runs through the book. Maya submits an equipment request, Ravi reviews it, and the surrounding software gradually grows from a small script into a production-shaped service. This continuity is intentional. It lets the reader see that the language used in a first program is still present in a larger system; what changes is the discipline around boundaries, responsibilities, failures, data, concurrency, deployment, and operational control.

The book therefore treats production thinking as part of Python learning rather than as an appendix added after the language has been taught. Code examples are kept small enough to inspect, while the surrounding notes make assumptions, verification levels, failure modes, and trade-offs visible. A working example is useful, but a reader should also be able to explain why it works, where it can fail, and what would need to change before the same idea is trusted in a real system.

The later chapters intentionally introduce more judgment than syntax. There is rarely one universally correct class structure, concurrency model, API shape, packaging workflow, or deployment strategy. The reader is instead encouraged to ask better engineering questions: What is the contract? Where is the boundary? Which invariant must remain true? Who owns the lifecycle? What failure is possible? What evidence would justify the added complexity? Those questions are more durable than any single framework or library version.

This is also why the book supports a two-pass learning path. A new programmer can follow the core route, build the mental model, and continue forward without mastering every advanced mechanism immediately. A more experienced developer can return to the deeper sections, failure cases, and extended labs when a real project gives those details context. Skipping an advanced section on the first reading is not a gap in learning; forcing abstraction before its purpose is understood usually is.

Most of all, this book is meant to be used actively. Read with a terminal open. Predict what a program will do before running it. Change one thing at a time. Keep failures long enough to understand them. Write tests that protect behavior rather than merely repeat implementation. Measure before optimizing. By the capstone, the goal is not simply to have seen a wide range of Python features, but to have connected them into a coherent way of thinking about software.

Python will continue to evolve, and the surrounding ecosystem will change even faster. The durable objective of this edition is therefore not memorization. It is to give the reader a foundation strong enough to learn new libraries, inspect unfamiliar systems, challenge weak assumptions, and make engineering decisions with greater clarity.

Team of Swatantra

SWATANTRA.ai

Table Of Contents

Copyright and edition notice

This book is a technical learning resource. Examples are intentionally small enough to run and inspect. Production notes describe risks and trade-offs but cannot replace project-specific architecture, security review, load testing, or operational procedures. In this edition, "production-oriented" means that an example illustrates a production-relevant boundary, control, or failure mode; it does not mean that the example has been deployed, load-tested, security-tested, or certified for an arbitrary production environment.

Who this book is for

This edition begins with a reader who may have installed Python five minutes ago. The early promise is modest: type a script, run it, change it, and understand the result. Only after that does the book introduce names and objects, design, reliability, concurrency, APIs, packaging, deployment, and security.

Experienced developers can use the early chapters as a fast review and begin deeper study at the object model, reliability, concurrency, or production sections. Technical leaders can use the design rules, failure modes, and capstone as review checklists.

- Primary reader: a developer who knows little or some Python and wants a serious progression.
- Secondary reader: a working developer correcting shallow or inaccurate mental models.
- Useful prerequisite: comfort with files, terminals, and basic programming ideas.
- Not assumed: prior knowledge of metaclasses, asyncio, HTTP services, containers, or packaging.

Learning path

Part	Focus	Reader outcome
I. Language Foundations	Execution, values, control flow, functions, collections, files and text	Write clear programs and understand ordinary Python behavior.
II. Python Design	Object model, OOP design, functions as objects, lazy iteration	Create abstractions whose behavior and failure modes remain visible.
III. Reliability and Concurrency	Algorithms, tests, logging, profiling, threads, processes, asyncio	Build code that is measurable, testable, and matched to its workload.
IV. Production Systems	Metaprogramming, APIs, packaging, deployment and security	Turn modules into controlled, installable, observable services.
V. Capstone	Employee equipment request service	Integrate domain design, persistence, HTTP contracts, testing, and operations.

> **The capstone is the thread that holds the book together: later chapters reuse earlier decisions instead of restarting with unrelated examples.**

Code verification policy

Code examples are verified at different levels depending on what they require. Self-contained Python examples were run under Python 3.13.5; examples that depend on external services, packages, credentials, network access, shell state, databases, or operating-system behavior were checked only to the level the environment allowed. Appendix A summarizes verification by category; the companion manifest records the status for each example.

Shell commands, SQL, TOML, Dockerfiles, and configuration examples are checked in their own formats rather than treated as Python. The four statuses below describe exactly what was verified.

- Executed: ran successfully in an isolated local process.
- Syntax-checked: parsed successfully but not run against external infrastructure.
- Format-reviewed: checked for structural consistency in its own language or configuration format.
- Conceptual: intentionally incomplete and labeled as such.

Companion verification repository. The release manifest records every code and configuration block and the method used to check it. A verified block is verified only to that stated level; it is not a claim that external infrastructure was exercised.

Reference notation. Version-sensitive and standards-sensitive passages use bracketed keys such as [PY-THREAD-314] and [HTTP-9110]. Full primary-source references appear in Appendix L.

How to use this book

Read with a terminal open. For each core example, predict one result, run it, and compare the output with your prediction. That loop builds a more useful mental model than passive reading.

New programmers should follow Part I in order, then continue through design, reliability, APIs, and the capstone. Experienced Python developers can skim Part I and enter at the object model, reliability, concurrency, or production sections. Chapter 15 is optional on a first reading.

> From Chapter 7 onward, use two passes when the material becomes dense: learn the main mechanism and complete the build exercise first; return to internals and edge cases when the capstone or real work gives them a purpose.

- Run examples in a clean virtual environment.
- Keep a short engineering notebook of predictions, failures, and corrections.
- Write tests for every lab, including one expected failure and one edge case.
- Record Python and dependency versions with the code.

Recommended reading routes

Reader	Suggested route	Best for
New to Python	Ch 1–14 → Ch 16–19; Ch 15 optional	A complete progression from first script to production service.
Experienced developer, new to Python	Ch 2 & Ch 4 → Ch 7–19	A fast language-model refresh followed by design and production topics.
Backend / API developer	Ch 7–14 → Ch 16–19	Design, reliability, concurrency, HTTP, persistence, and the capstone.
Architect / technical lead	Ch 8 → Ch 11–19	Boundaries, failure contracts, concurrency, APIs, releases, and operations.
Platform / DevOps reader	Ch 12 → Ch 16–19	Testing, observability, packaging, deployment, security, and release controls.

PART I - LANGUAGE FOUNDATIONS

From a reproducible interpreter to files, text, and explicit failure handling.

Chapter 1 - Your First Python Program

Install Python, run one small script, change it, and make sense of what happens.

Learning objectives

- Confirm that Python runs on your machine.
- Create and save a small .py file.
- Use names, input, output, and one decision.
- Read a simple error without panicking.

Python is a general-purpose programming language, but that description is not useful on your first day. Today, Python is simply a tool that can read a file of instructions and produce a result. You will make that happen before learning how the runtime works internally.

Production readiness comes later. For now, focus on one honest loop: write a line, run it, observe the result, and change it deliberately. That cycle matters more than memorizing terminology on the first day.

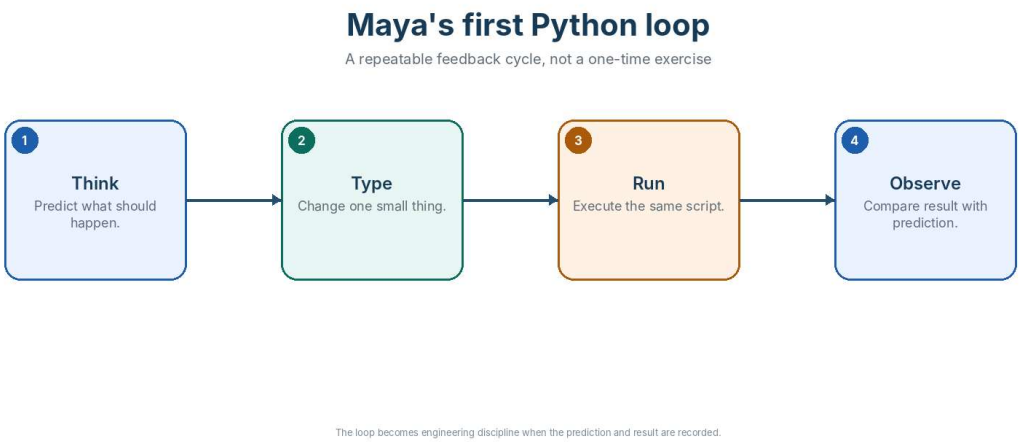

Figure 1.1 - The first programming loop is intentionally small.

1.1 Install Python and prove that it runs

Install Python from a trusted distribution for your operating system. Then open a terminal and ask Python for its version. You are not testing your memory; you are checking that the machine can find an interpreter.

On Windows, the command may be python or py. On macOS and Linux, python3 is common. Use the command that actually works on your machine and write it in your notes. The rest of the book assumes you can run a saved script from that terminal.

First checkpoint: run python --version (or py --version) and record the exact output.

Your first saved script

Create a file named request.py. Maya has joined the company and needs a laptop, so give the script one clear job: print that request. Save the file before running it.

```
# request.py
employee = "Maya"
item = "laptop"
print(f"{employee} requested a {item}")

# Terminal
python request.py
```

Change one value and run it again

Change item from "laptop" to "monitor" and run the file again. Nothing mysterious happened: the name item was bound to a different value before print used it. Chapter 2 gives the precise model. For now, notice that the program changed because you changed the value bound to item.

```
employee = "Maya"
item = "monitor"
print(f"{employee} requested a {item}")
```

Keep only three words for now: a name labels a value, an expression produces a value, and a statement performs an instruction. Appendix C holds the full reference.

1.2 Ask for input and produce output

A useful script can accept information instead of hard-coding every value. input() pauses and returns text. Store that text under a descriptive name, then use it in the message.

Run the program twice with different items. Predict the exact sentence before you press Enter. That prediction is the beginning of a working mental model.

```
employee = input("Employee name: ").strip()
item = input("Requested item: ").strip()
print(f"{employee} requested a {item}")
```

Try it twice: once with "laptop," once with "keyboard." Write the sentence you expect before each run.

1.3 Make one simple decision

Ravi reviews laptop requests, but keyboards can be approved automatically. An if statement lets the program choose one path or another. The indented lines belong to the decision above them.

Type the example exactly once. Then change the item and predict which message will appear. A wrong prediction is useful because it tells you what to inspect.

```
item = "laptop"

if item == "laptop":
  print("Send to Ravi for review")
else:
  print("Approve automatically")
```

```
OUTPUT
Send to Ravi for review
```

1.4 Repeat work with a loop

A loop repeats a block for each value. Here, the program processes three requested items. You do not need to memorize loop vocabulary yet. Follow the values in order and watch the same print statement run three times.

Change the list, rerun the script, and check whether the output order matches the input order.

```
items = ["laptop", "monitor", "keyboard"]

for item in items:
  print(f"Checking {item}")
```

1.5 Put the work inside a function

A function gives a piece of behavior a name. That makes the behavior reusable and easier to test later. The function below builds a sentence and returns it; print remains outside because displaying a result and calculating a result are different jobs.

A good editor configuration supports the workflow; it does not replace understanding. Learn how to run the same command from the terminal because continuous integration and production systems do not click an editor button.

- def introduces the function and its parameters.
- return gives a result back to the caller.

- Calling the function runs its body.
- Printing the returned value produces visible output.

```
def describe_request(employee, item):
    return f"{employee} requested a {item}"

message = describe_request("Maya", "laptop")
print(message)
```

1.6 Keep the startup line readable

As programs grow, keep startup code in a small main function. The familiar if __name__ == "__main__" line means “run this startup only when this file is executed directly.” You do not need the deeper import model yet; Chapter 4 returns to it.

For this first version, main returns nothing and the script simply prints a message. Appendix O explains process exit codes for automation and deployment.

```
def main():
    employee = "Maya"
    item = "laptop"
    print(f"{employee} requested a {item}")

if __name__ == "__main__":
    main()
```

OUTPUT
Maya requested a laptop

1.7 Read the last line of an error first

Sooner or later, a program fails. Good. A traceback is evidence, not a verdict. Start with the final line because it names the exception and usually explains the immediate problem. Then inspect the first line in your own file.

For a first exercise, deliberately misspell employee as employe. Run the script, read the NameError, fix the spelling, and run it again. That recovery matters more than memorizing every exception type.

- Read the exception type and message on the final line.
- Find the first frame that points to your file.
- Inspect the names and values used on that line.
- Change one thing, then run the same input again.
- Keep the fixed example as a reminder or a test.

1.8 Mini-project: Maya’s equipment request

Create request.py. Ask for an employee name, an item, and a quantity. Reject a blank name, use int() to convert the quantity, and print a short request summary. Keep the first version in one file. Structure can wait until the program works.

Run three cases: one valid request, one blank employee name, and one non-numeric quantity. Write down what failed before you change the code. In the next chapter, you will replace the rough conversion logic with a clearer boundary.

- You can create, save, and run a Python script.
- Names let later statements reuse values.
- input() returns text; print() produces visible output.
- if chooses a path; for repeats a block.

- A traceback gives you a place to begin.

Chapter 2 - Names, Values, Types, and Expressions

Python variables are names bound to objects, not boxes that permanently own a type.

Learning objectives

- Explain name binding and object identity.
- Use numeric, Boolean, string, and None values correctly.
- Convert external text explicitly.
- Distinguish equality from identity and mutability from rebinding.

An assignment binds a name to an object. It does not copy the object by default and it does not declare a permanent storage slot with a fixed type. A second name can refer to the same mutable object, so a change through either name becomes visible through the other.

This reference model explains aliasing, function arguments, object identity, and many surprising mutations. Use precise language: a name is rebound; an object is mutated; a new object is created. Saying that a variable "changed type" often hides which of those events actually occurred.

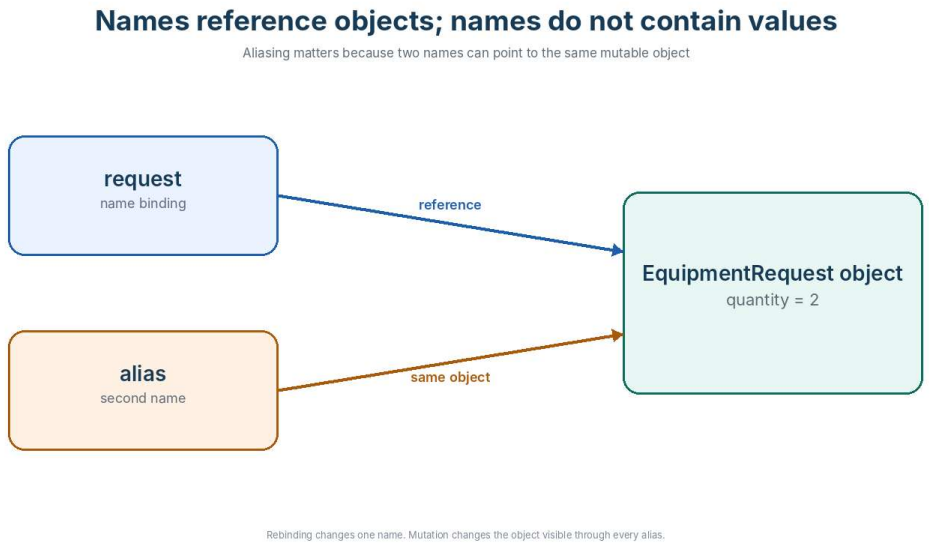

Figure 2.1 - Two names can refer to the same object.

Maya changes her request from one monitor to two. That tiny edit raises the first serious Python question: did the "variable" change, or did a name begin referring to another object? The distinction sounds fussy until shared mutable data causes a real bug.

2.1 Names, assignment, and fundamental types

A common beginner explanation says that a variable is a memory location that stores a value. That model is useful only as a first approximation. In Python, a variable is more accurately a name bound to an object. Assignment binds or rebinds the name; it does not permanently give the name a type.

```
age = 30
name = "Alice"
is_student = True
```

```
print(type(age).__name__)      # int
print(type(name).__name__)     # str
print(type(is_student).__name__)  # bool

age = "thirty"              # The name can be rebound to a different object.
print(type(age).__name__)      # str
```

Use descriptive snake_case names. A valid identifier begins with a letter or underscore, can contain letters, digits, and underscores, is case-sensitive, and cannot be a Python keyword.

- Good: employee_count, request_id, total_amount.
- Weak: x, data, temp when the meaning is not obvious.
- Invalid: 2nd_place, employee-name, class.
- Risky: list, str, id, input because they shadow useful built-ins.

Keep the four everyday values in view: integers for whole numbers, floats for approximate decimal values, strings for text, and booleans for decisions. The full type reference is in Appendix C.

Input, conversion, and validation

input() always returns text. Convert at the boundary and handle invalid input explicitly. Conversion functions such as int(), float(), str(), list(), tuple(), and set() create a value of the requested type or raise an exception when the conversion is not valid.

```
raw_quantity = "2"

try:
  quantity = int(raw_quantity)
except ValueError:
  print("Quantity must be a whole number")
else:
  if quantity <= 0:
    print("Quantity must be positive")
  else:
    print(f"Maya requested {quantity} monitors")
```

Operators and expression order

Arithmetic, comparison, logical, membership, identity, and assignment operators combine values into expressions. Parentheses are cheap documentation; use them when precedence is not immediately obvious.

```
price = 1200
quantity = 2
discount_rate = 0.10

subtotal = price * quantity
total = subtotal * (1 - discount_rate)
eligible = quantity > 0 and total <= 5000

print(total)
print(eligible)
print("laptop" in {"laptop", "monitor"})
print(total is None)
```

2.2 Integers, floating point, and Decimal

Python integers have arbitrary precision limited mainly by available memory. Floating-point values use binary representation and cannot exactly represent many decimal fractions. That is normal, not a Python defect. Equality checks involving calculated floats should consider an appropriate tolerance.

For money or decimal rules that require exact base-10 arithmetic, use decimal.Decimal and construct values from strings. A Decimal created from an existing float preserves the float approximation, which defeats the reason for choosing decimal arithmetic.

```
PYTHON
from decimal import Decimal
from math import isclose

print(0.1 + 0.2 == 0.3)
print(isclose(0.1 + 0.2, 0.3))
print(Decimal("0.1") + Decimal("0.2"))
```

OUTPUT
False
True
0.3

2.3 Booleans, truthiness, and None

Boolean expressions produce True or False, but conditional statements accept any object and apply truth-value testing. Empty strings and collections, zero numeric values, and None are false-like. Most other objects are true-like unless their class defines otherwise.

None represents the absence of a value. Compare it with is None because None is a singleton and identity is the intended question. Do not use a truthiness check when zero, an empty collection, or an empty string is a valid value that must be distinguished from absence.

```
PYTHON
def describe(quantity: int | None) -> str:
    if quantity is None:
        return "not supplied"
    if quantity == 0:
        return "none requested"
    return f"requested: {quantity}"

print(describe(None))
print(describe(0))
```

OUTPUT
not supplied
none requested

2.4 Strings, Unicode, and f-strings

Python strings are immutable sequences of Unicode code points. Source code can contain Unicode, but external files and network payloads are bytes that must be decoded with an encoding. UTF-8 is a common default for new systems, yet the correct encoding is part of the external contract rather than a guess.

Use f-strings for readable interpolation. Keep formatting separate from data storage: a formatted currency string is presentation, while a Decimal amount and currency code are data. Repeated

concatenation inside large loops can create unnecessary intermediate strings; collect fragments and join them when scale matters.

```python
employee = "Maya"
item = "monitor"
quantity = 2
message = f"{employee} requested {quantity} x {item}"
print(message)
```

OUTPUT
Maya requested 2 x monitor

2.5 Explicit type conversion at boundaries

Input from command-line arguments, environment variables, HTTP payloads, and text files normally begins as text. Convert at the boundary, validate immediately, and keep the internal representation consistent. Scattering int() and float() calls throughout business logic makes failure behavior difficult to locate.

Conversion can fail. int("ten") raises ValueError; int(3.9) truncates toward zero rather than rounding. A reliable parser defines acceptable formats and translates low-level conversion failures into a message meaningful to the caller.

A parsing bug becomes expensive when the same raw value is converted in six different layers. Convert at the boundary, reject it there, and let the rest of the program work with a real int.

```python
class InputError(ValueError):
    pass

def parse_monitor_quantity(raw: str) -> int:
    try:
        value = int(raw)
    except ValueError as exc:
        raise InputError("quantity must be a whole number") from exc
    if value <= 0:
        raise InputError("quantity must be positive")
    return value

print(parse_monitor_quantity("2"))
```

OUTPUT
2

2.6 Equality, identity, and hashing

The == operator asks whether two values are equivalent according to their type. The is operator asks whether two references point to the same object. Use is for None and for deliberate identity checks; use == for strings, numbers, collections, and value objects.

Hashable objects can be dictionary keys or set members. Their equality and hash must remain stable while stored. Immutable built-in values are usually safe keys. A mutable object whose hash changes after insertion can become impossible to find through the container assumptions it violated.

```python
a = [1, 2]
b = [1, 2]
c = a

print(a == b, a is b)
```

```
print(a == c, a is c)
```

OUTPUT
True False
True True

2.7 Mutability and copying

Lists, dictionaries, and sets are mutable; strings, integers, and tuples are immutable, although a tuple can contain a mutable object. Rebinding a name to a new value is not the same as mutating the object currently referenced by that name.

A shallow copy creates a new outer container while retaining references to nested objects. A deep copy recursively copies supported nested objects, but it can be expensive and can duplicate objects that should remain shared. Prefer explicit construction when the ownership model matters.

```
PYTHON
original = [[1], [2]]
shallow = original.copy()
shallow[0].append(99)

print(original)
print(shallow)
```

OUTPUT
[[1, 99], [2]]
[[1, 99], [2]]

2.8 Operators and expression evaluation

Expressions combine values, names, operators, and function calls to produce a result. Operator precedence determines grouping, but readable code should not make the reader reconstruct a complicated precedence table. Parentheses are cheap; misunderstood expressions are not.

Logical operators short-circuit. In a and b, b is evaluated only when a is truthy. In a or b, b is evaluated only when a is false-like. The operators return one of their operands rather than forcing a bool, which enables defaults but can also hide a valid zero or empty value.

Use parentheses when grouping is not obvious. Appendix C contains the complete operator table; the main chapter keeps only operators used by the running example.

2.9 Build: parse an equipment request

Build a parser for an equipment request containing employee_id, item_type, quantity, and optional budget. Accept raw strings, convert them once, reject invalid values, and return a dictionary or dataclass containing consistent internal types.

Add tests for a valid request, a non-numeric quantity, zero quantity, a missing optional budget, and a decimal budget. Demonstrate with == and is that two equal request values are not necessarily the same object.

- Names are bindings to objects.
- Runtime values have types; names can be rebound.
- Use Decimal for exact decimal rules and strings as its input.
- Use is None for absence and == for value equality.
- Convert and validate external text at boundaries.
- Mutability and aliasing are design concerns, not syntax trivia.

Chapter 3 - Control Flow and Comprehensions

Control flow should make decisions visible, terminate predictably, and avoid deeply nested logic.

Learning objectives

- Use indentation to define blocks.
- Write clear conditional branches and pattern matches.
- Choose for or while loops deliberately.
- Use comprehensions without hiding complex behavior.

Python uses indentation to define suites of statements. A colon introduces a block after constructs such as if, for, while, def, class, try, with, and match. Consistency is mandatory; four spaces per level is the conventional style.

Do not mix tabs and spaces. An editor can display invisible whitespace and insert spaces when the Tab key is pressed. A formatter can normalize layout, but it cannot decide whether the wrong block was indented. That remains a logic error.

```
PYTHON
for value in range(6):
    if value % 2 == 0:
        print(value)
```

```
OUTPUT
0
2
4
```

Ravi does not review every request. A keyboard may pass automatically; a laptop does not. Control flow is simply the code version of that decision, and the best version is the one another person can trace without guessing.

3.1 Conditions, loops, and a beginner execution trace

Control flow decides which statements run and how often they run. The safest way to learn it is to trace one small program line by line: identify the current values, evaluate the condition, and record which block executes.

```
score = 72

if score >= 80:
  grade = "A"
elif score >= 60:
  grade = "B"
else:
  grade = "C"

print(grade)
```

Conditions use truth values. Empty strings and collections, numeric zero, None, and False are falsy. Most other values are truthy. Prefer direct tests such as if items: rather than if len(items) > 0:.

The chapter explains if, for, while, break, and continue in the order they appear in Maya's request workflow. Appendix C keeps the compact reference table.

A complete loop example

```
requests = ["laptop", "", "monitor", "quit", "keyboard"]

for request in requests:
    if request == "quit":
        break
    if not request:
        continue
    print(f"Processing {request}")
else:
    print("Processed every request")

print("Loop ended")
```

The loop else block runs only if the loop finishes without break. It is valid Python, but a helper function that returns early is sometimes easier for a new team to read.

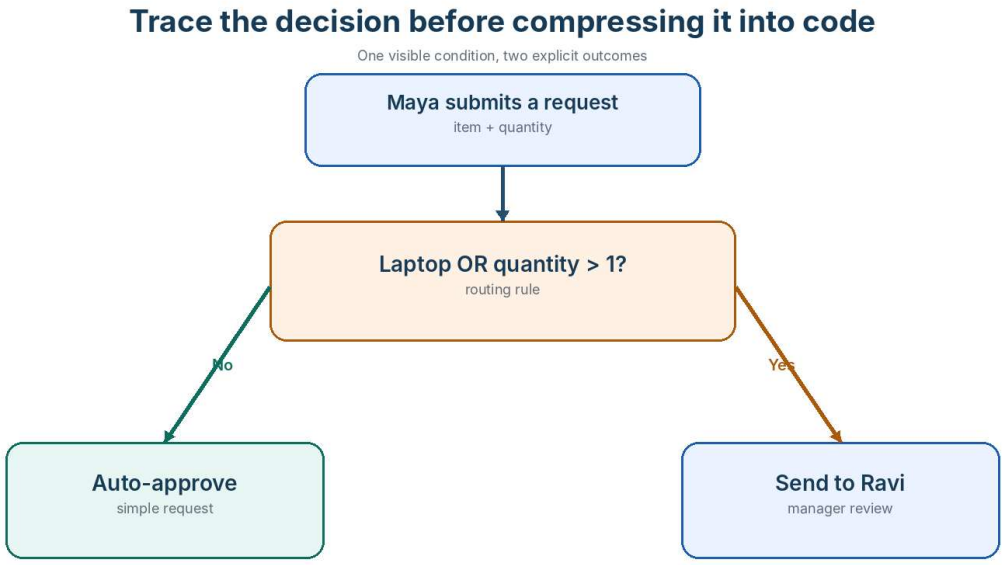

Figure 3.1 - Trace the decision before compressing it into code.

3.2 if, elif, and else

Conditional branches are checked from top to bottom. The first true condition runs and the remaining branches are skipped. Put specific cases before broad cases and use early returns when they reduce nesting.

An else branch is not always required. When every valid branch returns, a final explicit error can make unsupported states visible. Avoid long chains that compare the same value repeatedly when a mapping, polymorphic strategy, or match statement represents the decision more directly.

```
PYTHON
def route(quantity: int, item_type: str) -> str:
    if quantity <= 0:
        raise ValueError("quantity must be positive")
```

```
    if item_type == "laptop" or quantity > 1:
        return "manager_review"
    return "auto_approved"

print(route(1, "keyboard"))
```

OUTPUT
auto_approved

3.3 Structural pattern matching

The match statement compares a value against patterns. It is useful for structured alternatives such as command messages, tuples, and data shapes. It is not a faster or more sophisticated spelling of every if statement.

Patterns can bind names, so order matters. A broad capture pattern placed too early will match almost anything. Use a final case _ branch when unsupported input should be handled explicitly.

```
PYTHON
def describe_event(event: tuple[str, str]) -> str:
    match event:
        case ("submitted", request_id):
            return f"queue {request_id}"
        case ("approved", request_id):
            return f"fulfill {request_id}"
        case _:
            return "ignore"

print(describe_event(("approved", "REQ-9")))
```

OUTPUT
fulfill REQ-9

3.4 for loops, range, enumerate, and zip

A for loop asks an iterable for values and processes them one at a time. It is not limited to numeric counters. Use enumerate when both position and value are required, and zip when related iterables should be traversed together.

zip stops at the shortest input by default. If unequal lengths indicate a defect, use strict=True in supported Python versions so the mismatch raises instead of silently discarding extra data.

```
PYTHON
items = ["laptop", "monitor", "keyboard"]
priorities = [1, 2, 3]

for position, (item, priority) in enumerate(zip(items, priorities, strict=True), start=1):
    print(position, item, priority)
```

OUTPUT
1 laptop 1
2 monitor 2
3 keyboard 3

3.5 while loops and termination

Use a while loop when repetition depends on a changing condition rather than on a known iterable. The loop must contain a path that changes the condition, raises, returns, or breaks. Review that path explicitly; accidental infinite loops are often missing state transitions.

Polling loops need delays, deadlines, and cancellation behavior. A tight loop that repeatedly checks a resource consumes CPU while making no useful progress. In asynchronous code, use async waiting primitives rather than blocking sleep.

```python
attempt = 0
maximum = 3

while attempt < maximum:
    attempt += 1
    print(f"attempt {attempt}")
    if attempt == 2:
        break
```

OUTPUT
attempt 1
attempt 2

3.6 break, continue, pass, and loop else

break exits the nearest loop. continue skips the remaining statements in the current iteration. pass performs no operation and is useful only where syntax requires a statement. None of them should be described as pausing execution.

A loop else block runs only when the loop finishes without break. This can express a search whose failure is meaningful, but many teams find a helper function with an early return easier to read. Choose the form that makes the success and failure paths obvious.

A long loop full of continue statements usually hides the data-selection rule. If the exclusion can be stated before the loop, filter first; keep continue only when the decision genuinely belongs beside the work.

```python
values = [3, 7, 10, 12]
for value in values:
    if value % 5 == 0:
        found = value
        break
else:
    found = None

print(found)
```

OUTPUT
10

3.7 Comprehensions and generator expressions

A comprehension combines iteration, optional filtering, and construction. It is concise when the transformation is simple. It becomes hostile when it contains nested conditions, side effects, exception handling, or several levels of loops.

A list comprehension materializes a list. A generator expression produces values lazily. Choose based on the caller contract: repeated access and indexing need a collection; one-pass aggregation or streaming may benefit from a generator.

```python
requests = [
    {"id": "R1", "status": "approved"},
    {"id": "R2", "status": "submitted"},
]
approved_ids = [r["id"] for r in requests if r["status"] == "approved"]
```

```
print(approved_ids)
```

OUTPUT
['R1']

3.8 Reducing nested control flow

Deep nesting forces the reader to hold several conditions in memory. Guard clauses reject invalid or irrelevant cases early, leaving the main path less indented. Extracting a named predicate can also turn a complicated Boolean expression into a testable rule.

Do not flatten logic mechanically. Sometimes nesting accurately represents dependent decisions. The goal is to expose the business rule, not to minimize indentation at any cost.

```
PYTHON
def can_auto_approve(request: dict[str, object]) -> bool:
    if request.get("status") != "submitted":
        return False
    if request.get("item_type") == "laptop":
        return False
    return int(request.get("quantity", 0)) == 1
```

3.9 Build: a command-line request menu

Write a command-line menu that accepts submit, approve, reject, list, and quit commands. Use a loop with explicit termination, a small parser, and functions for each action. Reject commands with the wrong number of arguments rather than allowing an IndexError to leak.

Refactor the first version to reduce nesting. Add tests for an unknown command, an invalid quantity, a successful submit, and loop termination.

- Indentation defines program structure.
- Branches are evaluated in order.
- for loops consume iterables; while loops depend on a condition.
- break exits; continue skips; pass does nothing.
- Comprehensions are for simple transformations, not hidden workflows.
- Guard clauses and named predicates can expose the main path.

Chapter 4 - Functions, Scope, Modules, and Packages

Functions create contracts and change boundaries; modules turn those contracts into navigable code.

Learning objectives

- Define parameters and return values deliberately.
- Use packing and unpacking without hiding interfaces.
- Apply the LEGB lookup rule.
- Organize code into modules and packages with explicit imports.

A function is a reusable callable object created by def. Its signature communicates accepted inputs; its return value communicates the result. A function without an explicit return returns None. Printing a result is not the same as returning it.

Functions should usually perform one coherent responsibility at one level of abstraction. A thirty-line function is not automatically wrong, but a function that parses input, queries a database, applies policy, formats HTML, and sends email has several reasons to change.

PYTHON
```
def calculate_total(unit_price: float, quantity: int) -> float:
    if quantity <= 0:
        raise ValueError("quantity must be positive")
    return unit_price * quantity

total = calculate_total(249.5, 2)
print(total)
```

OUTPUT
```
499.0
```

The first script works, but Maya's request logic is now repeated in three places. Ravi asks for one fix. The team discovers three copies. Functions and modules exist to stop that kind of drift.

4.1 Defining, calling, and returning from functions

A function is a named, reusable unit of behavior. Parameters appear in the definition; arguments are the actual values supplied at the call site. A function should return information that callers need. Printing is an output effect and is not a substitute for a return value.

```
def calculate_total(price, quantity):
    """Return the total cost for a positive quantity."""
    if quantity <= 0:
        raise ValueError("quantity must be positive")
    return price * quantity

result = calculate_total(750, 2)
print(result)
```

- A function definition creates a callable object.
- A parameter is a name in the function signature; an argument is the value supplied at the call site.
- A return value gives information back to the caller.
- A side effect changes something outside the return value, such as printing, writing a file, or updating a database.
- Keep those roles separate in your head: the distinction makes testing and refactoring easier.

Defaults, packing, and beginner-safe signatures

Use defaults for genuinely optional behavior. Never use a mutable list or dictionary as a default unless persistent sharing is intentional. *args and **kwargs are useful tools, but they should not replace a readable business interface.

```
def add_tag(tag, tags=None):
    if tags is None:
        tags = []
    tags.append(tag)
    return tags

print(add_tag("urgent"))
print(add_tag("approved"))


def describe_request(request_id, *items, priority="normal", **metadata):
    return request_id, items, priority, metadata

print(describe_request("REQ-1", "laptop", "monitor", owner="E-10"))
```

From one file to modules

A module is a .py file. Importing a module executes its top-level code and binds the module object. Keep executable startup under if __name__ == "__main__": so the module can be imported by tests without accidentally running the program.

```
# calculator.py
def add(a, b):
    return a + b

# main.py
from calculator import add

if __name__ == "__main__":
    print(add(2, 3))
```

4.2 Parameter kinds

Python supports positional-only, positional-or-keyword, keyword-only, variable positional, and variable keyword parameters. Use the simplest signature that communicates the contract. Keyword-only parameters are valuable when several Boolean or numeric options would be ambiguous at the call site.

Default argument expressions are evaluated once when the function is defined. A mutable default therefore persists across calls. Use None as a sentinel or a default factory inside the function when each call needs a fresh list, dictionary, or set.

PYTHON

```
def create_request(employee_id: str, *, urgent: bool = False, tags: list[str] | None = None):
    safe_tags = [] if tags is None else list(tags)
    return {"employee_id": employee_id, "urgent": urgent, "tags": safe_tags}

print(create_request("E-7", urgent=True))
```

OUTPUT

```
{'employee_id': 'E-7', 'urgent': True, 'tags': []}
```

4.3 Packing and unpacking

*args packs extra positional arguments into a tuple; **kwargs packs extra keyword arguments into a dictionary. At a call site, * unpacks an iterable into positional arguments and ** unpacks a mapping into keyword arguments.

These tools are useful for adapters and decorators, but they can erase an interface. A business function defined only as *args, **kwargs forces readers and type checkers to discover its requirements elsewhere. Prefer explicit parameters unless forwarding is the actual responsibility.

PYTHON

```
def audit(event: str, *identifiers: str, **fields: object) -> dict[str, object]:
    return {"event": event, "identifiers": identifiers, **fields}

print(audit("approved", "REQ-1", actor="M-4"))
```

OUTPUT

```
{'event': 'approved', 'identifiers': ('REQ-1',), 'actor': 'M-4'}
```

4.4 Local, enclosing, global, and built-in scope

Python resolves an unqualified name through Local, Enclosing, Global, and Built-in scopes. Assignment normally creates or rebinds a local name. global and nonlocal change where assignment occurs, but frequent mutation of distant scope is a warning that state ownership is unclear.

A closure can retain an enclosing value after the outer function returns. That advanced behavior is covered later; the immediate rule is simpler: pass dependencies and return results rather than coordinating a program through writable globals.

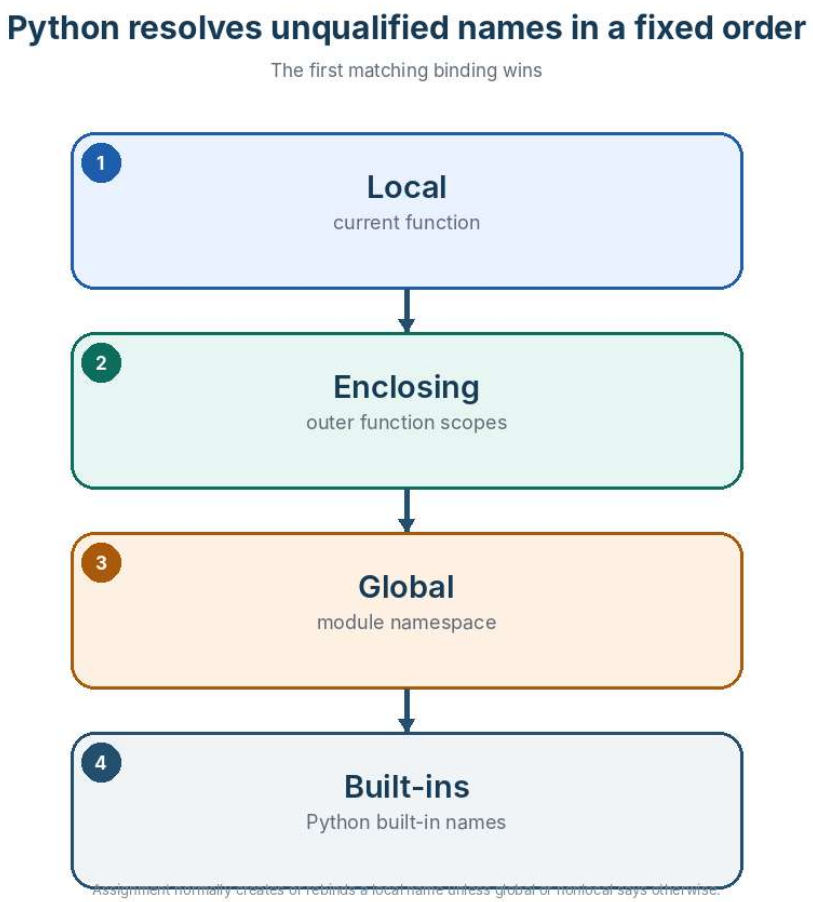

Figure 4.1 - Python resolves names through local, enclosing, global, and built-in scopes.

4.5 Functions are objects

A function can be stored in a variable, passed as an argument, returned from another function, or placed in a collection. This enables callbacks, strategies, validation pipelines, and decorators. The function retains metadata such as its name, annotations, defaults, and module.

Use a function as a strategy when behavior is small and stateless. Use an object when the behavior requires several operations, lifecycle management, or a substantial invariant. Both are ordinary Python design choices rather than competing ideologies.

PYTHON

```
def approve(request_id: str) -> str:
    return f"approved {request_id}"

def reject(request_id: str) -> str:
    return f"rejected {request_id}"

actions = {"approve": approve, "reject": reject}
print(actions["approve"]("REQ-8"))
```

OUTPUT

```
approved REQ-8
```

4.6 Docstrings, annotations, and contracts

A docstring explains purpose, assumptions, important failure behavior, and non-obvious constraints. Repeating the function name in a sentence adds little. Type annotations improve editor support and

static analysis but do not enforce values at runtime unless a framework or explicit validator interprets them.

A useful contract includes more than types: whether a function mutates input, whether it performs I/O, which exceptions are expected, whether it is idempotent, and whether the return is reusable or one-shot.

PYTHON
```
def normalize_item_type(raw: str) -> str:
    """Return a canonical item code; raise ValueError for a blank value."""
    normalized = raw.strip().lower().replace(" ", "_")
    if not normalized:
        raise ValueError("item type is required")
    return normalized
```

4.7 Modules and imports

A module is a Python file loaded as a module object. On the first normal import, Python executes the module's top-level code and caches the module object in sys.modules; later normal imports usually reuse that cached object. Explicit reloads can execute module-level code again. Expensive work, network calls, and environment validation at import time make testing and startup fragile.

Prefer explicit imports and stable module boundaries. from module import * hides where names came from and makes refactoring risky. Circular imports usually indicate that modules own responsibilities at the wrong level or depend on concrete implementations in both directions.

PYTHON
```
# pricing.py
def calculate(unit_price: float, quantity: int) -> float:
    return unit_price * quantity

# app.py
from pricing import calculate

print(calculate(50, 3))
```

OUTPUT
```
150
```

4.8 Packages and __init__.py

A package groups modules under a namespace. A regular package normally contains __init__.py; namespace packages are an advanced alternative for distributing one logical package across locations. The __init__.py file can be empty, expose a deliberate public API, or perform lightweight package initialization.

Do not turn __init__.py into a hidden application bootstrap. Imports should remain predictable. A docstring such as "HTTP route modules" documents the package for humans; tools do not consult that sentence to decide whether every import is allowed.

File	Responsibility
src/equipment/domain.py	Domain values and state transitions
src/equipment/service.py	Application orchestration
src/equipment/repository.py	Persistence interfaces and implementations
src/equipment/api.py	HTTP adapter
tests/	Behavior and boundary tests

4.9 Dependency direction and import hygiene

Place stable business rules where they do not import FastAPI, database drivers, or deployment code. Infrastructure can depend on the domain; the domain should not depend on infrastructure. This direction makes rules testable without starting a server or database.

Import cycles are often the first visible symptom of a broken dependency direction. Moving shared types into a neutral module can help, but creating a giant common.py file only relocates the problem. Assign each concept a clear owner.

> **A package boundary helps only when it matches ownership. Splitting files without changing dependency direction produces more folders, not better architecture.**

4.10 Build: split the request program into modules

Split the command-line equipment program into parser.py, domain.py, service.py, and main.py. Keep parsing at the boundary, return domain values from functions, and ensure domain.py imports no editor, HTTP, or database packages.

Add unit tests for the pure rules and one integration-style test that calls main with a controlled argument list. Verify that importing the package does not print, open files, or connect to services.

- Functions expose input, output, mutation, and failure contracts.
- Mutable defaults persist across calls.
- LEGB explains name lookup; global mutation should be rare.
- Functions are first-class objects.
- Modules execute top-level code when imported.
- Packages should make ownership and dependency direction easier to see.

Chapter 5 - Core Collections and Data Modeling

Choose collections by the operations and invariants the program needs, not by which syntax is most familiar.

Learning objectives

- Use lists, tuples, dictionaries, and sets deliberately.
- Reason about mutability and copying.
- Write readable comprehensions.
- Model records with dataclasses when a raw collection loses meaning.

A list stores an ordered sequence of references and supports indexing, slicing, append, extension, insertion, removal, and sorting. Appending at the right end is efficient; inserting or removing near the beginning requires shifting references and becomes costly for large lists.

A list can hold mixed types, but production code is usually clearer when every element follows one conceptual contract. A list of unrelated values is often an unnamed record that should become a dataclass, tuple, or dictionary with explicit fields.

PYTHON

```
items = ["laptop", "monitor"]
items.append("keyboard")
items[1] = "27-inch monitor"
print(items)
print(items[:2])
```

OUTPUT

```
['laptop', '27-inch monitor', 'keyboard']
['laptop', '27-inch monitor']
```

By Friday, Ravi has a list of requests, a set of approved item types, and a dictionary indexed by request ID. They are not interchangeable containers. Each one makes a different operation cheap and a different mistake likely.

5.1 Lists, tuples, dictionaries, and sets

Python's four core collection types solve different problems. Learn their operations before memorizing methods. A collection should communicate what the data means and how it will be accessed.

Read the examples first. Use Appendix C when you need the side-by-side collection reference.

```
items = ["laptop", "monitor"]
items.append("keyboard")
items[0] = "business laptop"
print(items[0:2])

request = ("REQ-1", "submitted")
print(request[1])

employee = {"id": "E-10", "department": "IT"}
employee["location"] = "Bengaluru"
print(employee.get("manager", "not assigned"))
```

```
requested_types = {"laptop", "monitor", "laptop"}
print(requested_types)
print("monitor" in requested_types)
```

Indexing, slicing, aliasing, and copying

Indexing selects one item; slicing creates a new sequence over a range. Assignment does not copy an object. Two names can refer to the same mutable list, so a change through one name appears through the other.

```
original = [1, 2, 3]
alias = original
shallow_copy = original.copy()

alias.append(4)
print(original)     # [1, 2, 3, 4]
print(shallow_copy)  # [1, 2, 3]

matrix = [[1, 2], [3, 4]]
copy_of_matrix = matrix.copy()
matrix[0].append(99)
print(copy_of_matrix)  # Inner lists are still shared.
```

Choosing a clearer data model

A dictionary is useful at a parsing boundary, but repeated string keys and undocumented shapes become fragile. When a record has stable fields and business meaning, a dataclass can make the contract visible.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class EquipmentItem:
    item_type: str
    quantity: int

item = EquipmentItem("monitor", 2)
print(item.item_type, item.quantity)
```

Figure 5.1 - Start from the operation, not the familiar syntax.

5.2 Tuples: fixed structure and immutable references

A tuple is an ordered sequence whose element references cannot be replaced after creation. It is useful for small fixed records, multiple return values, and hashable composite keys when all members are hashable.

Tuple immutability is shallow. A tuple can refer to a list that later changes. Parentheses do not create a one-element tuple by themselves; the trailing comma does. When field meaning matters, a named tuple or frozen dataclass is often clearer than positional indexes.

PYTHON
```
point = (10, 20)
single = ("monitor",)
mutable_inside = ([1, 2], "notes")
mutable_inside[0].append(3)
print(point, single, mutable_inside)
```

5.3 Dictionaries: key-based lookup

A dictionary maps hashable keys to values and preserves insertion order in modern Python. Lookup, insertion, and deletion are average constant time, subject to hashing behavior and implementation limits. Keys express meaning more clearly than numeric positions.

Access with mapping[key] when absence is exceptional. Use get when absence is an expected branch and choose the default carefully. setdefault can be convenient, but defaultdict or an explicit branch may make repeated grouping logic clearer.

PYTHON
```
request = {"id": "REQ-1", "status": "submitted"}
request["status"] = "approved"
request["quantity"] = 1
print(request.get("status"))
print("employee_id" in request)
```

OUTPUT
```
approved
False
```

5.4 Sets: uniqueness and membership

A set stores unique hashable elements and supports union, intersection, difference, and average constant-time membership. Sets do not support positional indexing. Their iteration order must not be used as a business ordering contract.

Use a set when uniqueness or membership is the primary operation. Converting a list to a set removes duplicates but also discards duplicate counts and may alter order. That is a semantic change, not merely an optimization.

PYTHON
```
requested = {"laptop", "monitor"}
available = {"monitor", "keyboard", "mouse"}
print(requested & available)
print(requested - available)
```

5.5 Comprehensions for construction

List, set, and dictionary comprehensions build collections from iterables. They are clearest when the expression and filter fit one mental step. Move validation, logging, exception handling, and side effects into named functions before using them in a comprehension.

Nested comprehensions are not forbidden, but readability drops quickly. A normal loop can communicate intermediate names, branches, and failure handling better than a compressed expression.

```python
quantities = {"laptop": 1, "monitor": 2, "mouse": 0}
positive = {item: qty for item, qty in quantities.items() if qty > 0}
labels = [f"{item}:{qty}" for item, qty in positive.items()]
print(positive)
print(labels)
```

5.6 Sorting, keys, and stable order

sorted returns a new list; list.sort mutates an existing list and returns None. Both accept a key function. Python sorting is stable, so records with equal keys preserve their previous relative order. Stable sorting enables multi-stage sorts when necessary.

Do not implement comparison methods solely to sort one report. A key function keeps report-specific ordering outside the domain value. Define value ordering only when the ordering is intrinsic and consistent across callers.

```python
requests = [
    {"id": "R2", "priority": 2},
    {"id": "R1", "priority": 1},
]
ordered = sorted(requests, key=lambda r: (r["priority"], r["id"]))
print(ordered)
```

5.7 Copying and nested mutation

Assignment shares a reference. copy creates a shallow outer copy. copy.deepcopy recursively duplicates supported objects, tracks cycles, and can invoke custom copy behavior. None of these choices answers the design question of who owns mutable state.

Prefer immutable values at boundaries and explicit update functions when concurrent or long-lived code would otherwise share nested mutation. A defensive copy is useful only when the copying policy is understood and consistently applied.

```python
from copy import deepcopy

original = {"items": [{"type": "monitor"}]}
clone = deepcopy(original)
clone["items"][0]["type"] = "keyboard"
print(original)
print(clone)
```

5.8 Dataclasses for named records

When a dictionary or tuple represents one domain concept, a dataclass can expose names, defaults, equality, representation, and optional immutability. It does not automatically validate fields or make nested values immutable.

Use __post_init__ for invariants that must hold after generated initialization. frozen=True prevents normal assignment and can support hashing when fields are hashable, but it should represent a genuinely immutable value rather than serve as decoration.

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class RequestLine:
```

```
    item_type: str
    quantity: int

    def __post_init__(self) -> None:
        if self.quantity <= 0:
            raise ValueError("quantity must be positive")

print(RequestLine("monitor", 2))
```

5.9 A practical selection rule

Choose the structure from the dominant operation, then confirm the choice with representative data:

- Use a list for an ordered mutable sequence.
- Use a tuple or frozen dataclass for a small fixed record or composite key.
- Use a dictionary for lookup by a meaningful key.
- Use a set when uniqueness or membership is the main question.
- Use deque for queue work at both ends and Counter for frequencies.
- Appendix C keeps the side-by-side reference; the chapter keeps the decision.

> **Complexity tells you where a design may stop scaling. It does not tell you whether the current program is slow. Measure representative data before trading clarity for speed.**

5.10 Build: model and report equipment requests

Read a sequence of equipment-request dictionaries. Validate them into frozen RequestLine objects, count item types, group request IDs by employee, and produce a priority-sorted report without mutating the input collection.

Add tests for duplicates, an invalid quantity, missing keys, and stable ordering. Explain why each chosen collection fits the operations performed.

- Lists are ordered and mutable.
- Tuples fix element references but can contain mutable objects.
- Dictionaries model key-based lookup.
- Sets model uniqueness and membership.
- Comprehensions should remain a single readable transformation.
- Dataclasses name records but do not remove the need for invariants.

Chapter 6 - Files, Text, Exceptions, and Regular Expressions

External data is where encodings, malformed input, missing files, and incomplete assumptions become visible.

Learning objectives

- Use pathlib and context managers for file access.
- Read and write text with explicit encodings.
- Design exception handling as part of an interface.
- Use string methods before reaching for regular expressions.

pathlib.Path represents filesystem paths without manually joining separators. A path object does not guarantee that a file exists or that the process has permission to access it. Those conditions are checked when an operation occurs.

Use relative paths only when the working-directory contract is clear. Library code should not assume that the current directory is the package directory. Accept a path, derive it from configuration, or use package-resource APIs for bundled data.

PYTHON
```
from pathlib import Path

base = Path("data")
request_file = base / "requests.json"
print(request_file.as_posix())
```

OUTPUT
```
data/requests.json
```

The first production-shaped failure is rarely a clever algorithm. It is a missing file, malformed JSON, an unexpected encoding, or a value that looks valid until the business rule checks it.

6.1 Strings, paths, files, and first exception handling

Strings are immutable Unicode text. Use methods such as strip(), split(), join(), lower(), upper(), startswith(), and replace() for straightforward transformations before reaching for regular expressions.

```
raw = "  Laptop, Monitor, Keyboard  "
items = [part.strip().lower() for part in raw.split(",")]
print(items)
print(" | ".join(items))
print(raw.strip().startswith("Laptop"))
```

Use pathlib.Path for paths instead of manually joining strings. Open text files with an explicit encoding and a context manager so the file is closed even when an exception occurs.

```
from pathlib import Path

path = Path("equipment.txt")
path.write_text("laptop\nmonitor\n", encoding="utf-8")

try:
```

```
    content = path.read_text(encoding="utf-8")
except OSError as exc:
    print(f"Could not read file: {exc}")
else:
    print(content.splitlines())
finally:
    path.unlink(missing_ok=True)
```

The try / except / else / finally structure

Block	When it runs	Use
try	First	Code that may raise a known failure.
except	When a matching exception occurs	Recover, translate, or report.
else	Only when try succeeds	Success-path work that should not be caught.
finally	Whether success or failure occurs	Cleanup that must always run.

Regular expressions after string methods

A regular expression describes a text pattern. Use raw strings for patterns. Keep patterns small, name captured groups, and test rejection cases. For simple prefix, split, or replacement tasks, ordinary string methods are usually clearer.

```
import re

pattern = re.compile(r"^(?P<prefix>REQ)-(?P<number>\d{4})$")
match = pattern.fullmatch("REQ-0042")

if match:
    print(match.group("number"))
else:
    print("Invalid request ID")
```

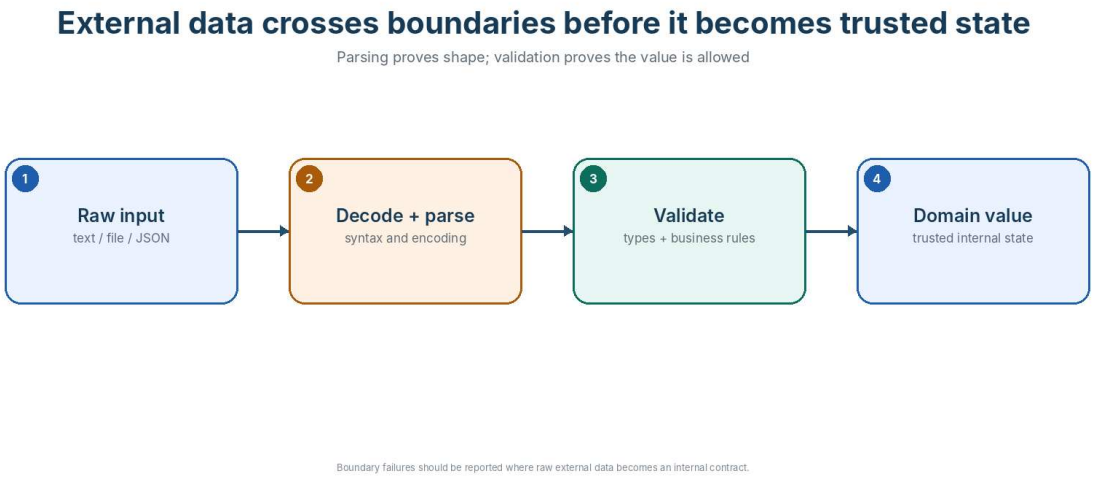

Figure 6.1 - External text becomes trusted internal state in stages.

6.2 Context managers for deterministic cleanup

The with statement guarantees that the file object is closed when the block exits, including when an exception occurs. This deterministic lifecycle is clearer and safer than relying on garbage collection or remembering a close call in every branch.

Specify the encoding for text. newline handling can also matter for CSV and cross-platform output. Use binary mode when the content is bytes rather than text.

PYTHON
```
from pathlib import Path

path = Path("example.txt")
with path.open("w", encoding="utf-8") as stream:
    stream.write("monitor\n")

with path.open("r", encoding="utf-8") as stream:
    print(stream.read().strip())

path.unlink()
```

OUTPUT
```
monitor
```

6.3 JSON, CSV, and schema boundaries

JSON represents objects, arrays, strings, numbers, booleans, and null. It does not preserve Python tuples, Decimal values, datetime objects, or custom classes without an explicit serialization policy. Loading JSON proves only that the syntax is valid; business validation is still required.

CSV is a tabular interchange format with quoting and dialect rules. Use the csv module rather than splitting lines by commas. A comma can legally appear inside a quoted field.

PYTHON
```
import json

payload = '{"employee_id": "E-1", "quantity": 2}'
data = json.loads(payload)
if not isinstance(data.get("quantity"), int):
    raise ValueError("quantity must be an integer")
print(data["employee_id"])
```

OUTPUT
```
E-1
```

6.4 try, except, else, and finally

Catch an exception only when the code can recover, add context, translate it into a meaningful error, or guarantee cleanup. Catching Exception and continuing with a default value often turns a visible defect into plausible incorrect output.

The else block runs when the try block succeeds and can keep unrelated code out of the protected region. finally runs whether the operation succeeds or fails, but context managers are usually the better resource-cleanup abstraction.

Catching Exception to make a traceback disappear usually converts a visible defect into a vague downstream failure. Catch only what this boundary can recover from or translate.

PYTHON
```
def parse_port(raw: str) -> int:
    try:
        port = int(raw)
    except ValueError as exc:
        raise ValueError("port must be numeric") from exc
    else:
        if not 1 <= port <= 65535:
            raise ValueError("port is out of range")
        return port

print(parse_port("8000"))
```

OUTPUT
```
8000
```

6.5 Custom exceptions and chaining

A custom exception should express a meaningful condition that callers can handle differently. Do not create a new class merely to rename every built-in error. Translate at a boundary where the low-level detail would otherwise leak into a higher-level contract.

Use raise NewError(...) from exc to preserve the original cause. The traceback then shows both what the infrastructure reported and what the application concluded.

PYTHON
```
class RequestFileError(Exception):
    pass

def load_text(path):
    try:
        return path.read_text(encoding="utf-8")
    except OSError as exc:
        raise RequestFileError(f"cannot read {path}") from exc
```

6.6 Start with string methods

Many text tasks need no regular expression. strip, split, partition, startswith, endswith, replace, and casefold are easier to read and often easier to test. Use the least powerful mechanism that accurately expresses the rule.

Text normalization is a business decision. Lowercasing an identifier, removing punctuation, or collapsing whitespace can change meaning. Preserve the original input when audit or user feedback may require it.

PYTHON
```
raw = "  Laptop Request : urgent  "
label, separator, value = raw.strip().partition(":")
print(label.strip().casefold())
print(value.strip())
```

OUTPUT
```
laptop request
urgent
```

6.7 Regular expressions as explicit patterns

The re module searches, matches, splits, and substitutes text using regular-expression patterns. Use raw string literals for patterns so backslashes are not interpreted twice. fullmatch is appropriate when the entire input must satisfy a format; search finds a match anywhere.

Compile a pattern when it is reused or when giving it a name improves readability. Named groups make extracted fields self-documenting.

PYTHON
```
import re

request_id_pattern = re.compile(r"(?P<prefix>REQ)-(?P<number>\d{4})")
match = request_id_pattern.fullmatch("REQ-0042")
if match is None:
    raise ValueError("invalid request id")
print(match.group("number"))
```

OUTPUT
```
0042
```

6.8 Regex failure modes and security

A complicated pattern can be more difficult to maintain than a parser. Some patterns exhibit catastrophic backtracking on carefully chosen input, consuming excessive CPU. Bound input size,

avoid ambiguous nested repetition, and use time-limited or alternative parsing approaches when untrusted text can be large.

Regex validates textual shape, not domain truth. A date-shaped string can still describe an impossible date; an email-shaped string does not prove that an account exists.

- Do not copy a pattern without understanding every group and quantifier.
- Prefer fullmatch for whole-field validation.
- Keep input length bounded at external boundaries.
- Follow structural matching with domain validation.
- Add tests for near-matches and adversarial long input.

6.9 Build: validate a JSON Lines request file

Read newline-delimited JSON request records from a UTF-8 file. Skip blank lines, preserve the source line number, parse JSON, validate required fields, and collect errors without losing successful records. Validate request IDs with a compiled fullmatch pattern.

Write a summary file atomically by first writing to a temporary path on the same filesystem and then replacing the target with os.replace. This makes the name replacement atomic, but does not by itself guarantee crash durability. Test a missing file, invalid UTF-8, malformed JSON, an invalid request ID, and successful cleanup.

- Use Path and explicit encodings.
- Context managers own resource cleanup.
- Serialization syntax and domain validation are separate.
- Exceptions are part of the caller contract.
- String methods are often clearer than regex.
- Regex patterns need performance and adversarial-input tests.

BRIDGE - FROM SCRIPTS TO SYSTEMS

After Chapter 6, syntax stops being the main difficulty.

The request program can already read input, branch, loop, use files, and report failures. The next problems are ownership, change, concurrency, and operation: where behavior belongs, how dependencies are controlled, and what happens when the system is no longer running only on one developer's machine.

> Parts II–IV introduce those decisions only when the equipment-request service needs them. Read the main path first; return to internals and edge cases when the capstone or a real system makes them concrete.

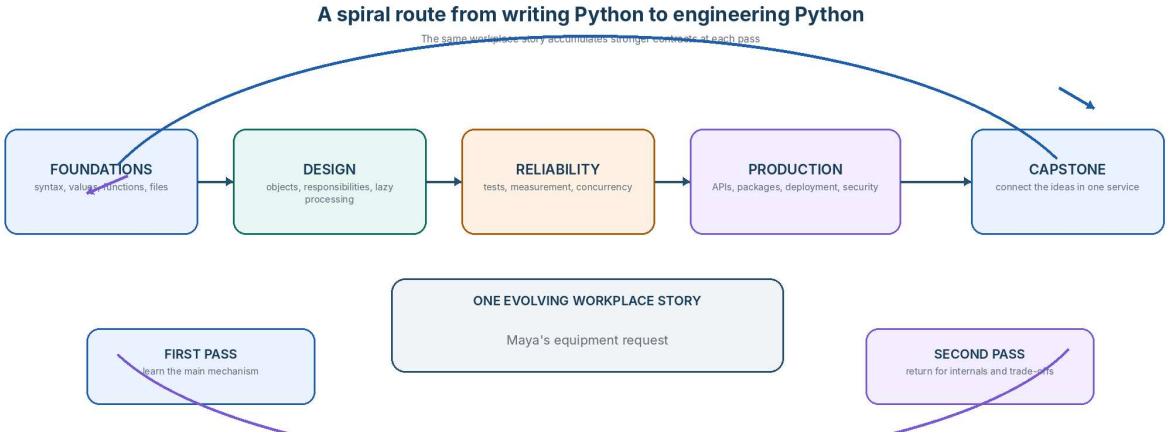

Figure Bridge.1 - The advanced half of the book is a spiral, not a staircase.

Five ideas that recur throughout the book

- **Contract:** what a caller can rely on: inputs, outputs, effects, and failures.
- **Boundary:** the place where one kind of responsibility or data becomes another.
- **Invariant:** a rule that must remain true while an object or workflow changes.
- **Lifecycle:** how something is created, used, stopped, cleaned up, or replaced.
- **Trade-off:** a choice that improves one quality while increasing another cost.

Keep the request workflow in view

Maya submits an equipment request, Ravi reviews it, and the system stores the decision and explains failures. Use that workflow as the test for every advanced mechanism: if a concept does not make the path clearer, safer, more testable, or easier to operate, it probably does not belong in the first implementation.

PART II - PYTHON DESIGN

The object model, responsibilities, functions as objects, and lazy resource-safe processing.

Chapter 7 - The Python Object Model

Understand how instances are created, initialized, and constrained.

Learning objectives

- distinguish classes from instances
- explain the roles of __new__ and __init__
- use class and instance attributes correctly
- control attributes with properties

Maya's request is now represented by classes rather than loose values. That creates a more useful set of questions: when is an instance created, where does its state live, and how can invalid state be blocked without hiding behavior?

7.1 Classes and instances

A class defines behavior and usually establishes the valid state of its instances. An instance is a concrete object created from that class. Calling a class normally triggers two different operations: Python asks __new__ to create the instance, then calls __init__ to initialize the instance that was returned.

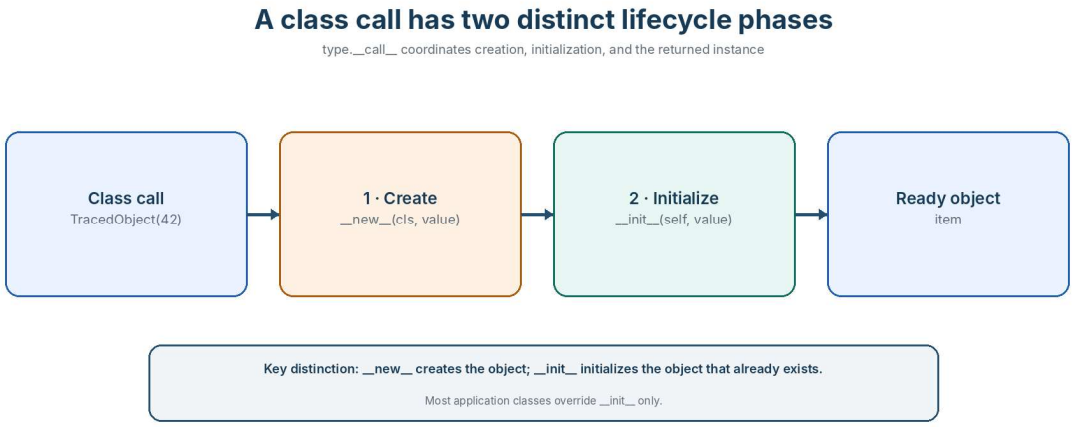

Figure 7.1 - A class call separates instance creation from initialization.

Correction: __init__ is commonly called a constructor in informal tutorials, but that description is imprecise. __new__ performs instance creation; __init__ performs initialization. Most application code only needs __init__, but the distinction matters when studying immutable types, metaclasses, or custom object creation.

Example 7.1 - Object creation: __new__ then __init__

```
PYTHON
class TracedObject:
    def __new__(cls, value):
        print("__new__ creates the instance")
        return super().__new__(cls)

    def __init__(self, value):
        print("__init__ initializes the instance")
        self.value = value
```

```
item = TracedObject(42)
print(item.value)
```

```
OUTPUT
Observed output:
__new__ creates the instance
__init__ initializes the instance
42
```

7.2 Attributes and lookup

Instance attributes normally live on each object. Class attributes live on the class and are visible through instances unless an instance attribute shadows them. Mutating a shared class attribute can affect every instance that still resolves that name through the class.

Example 7.2 - Class and instance attributes

```
PYTHON
class Device:
    category = "equipment"

    def __init__(self, name):
        self.name = name

laptop = Device("Laptop")
monitor = Device("Monitor")
Device.category = "office equipment"

print(laptop.name)
print(monitor.name)
print(laptop.category)
```

```
OUTPUT
Observed output:
Laptop
Monitor
office equipment
```

7.3 Properties and invariants

Python does not enforce private fields in the same way as some languages. A single leading underscore communicates that an attribute is internal. A double leading underscore triggers name mangling, which reduces accidental clashes but is not a security mechanism. Use properties when assignment must validate or transform a value.

Treat property access as cheap, local work. If reading an attribute can open a socket, hit a database, or wait seconds, expose that cost as a method instead.

Example 7.3 - Validated properties

```
PYTHON
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("temperature is below absolute zero")
        self._celsius = float(value)

reading = Temperature(24)
print(reading.celsius)
```

OUTPUT
Observed output:
24.0

7.4 Lifecycle, lookup, and restraint

Calling a class normally uses __new__ to create the instance and __init__ to initialize it. Attribute access can then involve descriptors, instance state, class attributes, and the method resolution order. You rarely need to customize creation, but knowing this path explains immutable subclasses, properties, shared class state, and surprising lookup behavior.

Keep the machinery proportional to the problem: put per-instance mutable state in __init__ or a default factory, use properties for local invariants, rely on context managers rather than __del__ for deterministic cleanup, and add __slots__ only after a measured memory need. When lookup is surprising, inspect type(obj), obj.__dict__, Class.__dict__, and Class.__mro__.

Worked example: a value object with a visible invariant

This example is small on purpose. Follow object creation once, then decide which invariant belongs in the type and which belongs at an external boundary.

PYTHON
```
class Temperature:
    __slots__ = ("_celsius",)

    def __new__(cls, celsius: float):
        instance = super().__new__(cls)
        return instance

    def __init__(self, celsius: float):
        self.celsius = celsius

    @property
    def celsius(self) -> float:
        return self._celsius

    @celsius.setter
    def celsius(self, value: float) -> None:
        value = float(value)
        if value < -273.15:
            raise ValueError("temperature is below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self) -> float:
        return self.celsius * 9 / 5 + 32

t = Temperature(25)
print(t.celsius, t.fahrenheit)
```

Expected result: 25.0 77.0

Pause before extending it: write down what __new__, __init__, and the property each guarantee.

Trade-offs and failure modes

The recurring object-model failures come from confusing creation, identity, shared state, and hidden cost. Keep properties cheap and local; use == for value semantics and is for identity; and remember that value equality affects hashing, so mutable value objects should not become unstable dictionary keys or set members.

- Using a mutable class attribute for per-instance state.
- Using is for string, numeric, or other value comparison.
- Hiding slow or failure-prone I/O inside a property.

- Adding __slots__ before measuring a real memory problem.

Build: a Money value you can defend

Build the smallest version first. Then inspect identity, equality, hashing, and mutation one at a time. Do not add __slots__ until you have measured a reason.

1. Create a Money value object with amount and currency fields. Reject unsupported currencies and non-finite numeric values.
2. Implement __repr__ for debugging and __eq__ for value comparison. Decide deliberately whether the object should be immutable and hashable.
3. Write a small script that creates two equal values and compares them with both == and is.
4. Inspect Money.__mro__, Money.__dict__, and an instance's available attributes. Record what changes if __slots__ is added.
5. Add tests for valid construction, invalid construction, equality, and attempted mutation.

What to keep from the exercise

- Source and tests
- A note on equality and hashability
- One measured comparison with and without slots
- The invalid cases you deliberately exercised
- Your final design decision

Practice

6. Modify Example 7.3 so that celsius accepts only int or float values.
7. Create a class attribute named category and demonstrate how an instance attribute can shadow it.
8. Explain why deleting a required business attribute is usually worse than representing an explicit state such as None.

Chapter 8 - Object-Oriented Design

Organize responsibilities so changes stay local.

Learning objectives

- apply polymorphism
- choose composition or inheritance deliberately
- use abstract base classes without overengineering
- recognize fragile class hierarchies

The request service now has several collaborators: approval policy, notification, and persistence. The design problem is no longer how to define a class; it is how to keep a change in one responsibility from rippling through the rest.

8.1 Encapsulation, abstraction, and polymorphism

Encapsulation groups state with the operations that preserve it. Abstraction exposes what a component guarantees while hiding unnecessary implementation detail. Polymorphism lets different objects satisfy the same operation. In Python this often works through duck typing, abstract base classes, or structural typing.

Example 8.1 - Polymorphism through a shared interface

PYTHON

```
from abc import ABC, abstractmethod

class Notifier(ABC):
    @abstractmethod
    def send(self, message):
        raise NotImplementedError

class EmailNotifier(Notifier):
    def send(self, message):
        return f"Email: {message}"

class ConsoleNotifier(Notifier):
    def send(self, message):
        return f"Console: {message}"

def notify(notifier, message):
    print(notifier.send(message))

notify(EmailNotifier(), "approved")
notify(ConsoleNotifier(), "approved")
```

OUTPUT

```
Observed output:
Email: approved
Console: approved
```

8.2 Inheritance versus composition

Inheritance creates an is-a relationship and couples a child class to a parent implementation. Composition creates a has-a relationship and often keeps independent behaviors replaceable. Use inheritance when the subtype genuinely satisfies the parent contract. Use composition when behaviors need to vary independently.

Example 8.2 - Prefer composition when behavior varies independently

PYTHON

```
class PercentageDiscount:
    def __init__(self, rate):
        self.rate = rate

    def apply(self, amount):
        return amount * (1 - self.rate)

class Order:
    def __init__(self, total, discount_policy):
        self.total = total
        self.discount_policy = discount_policy

    def final_total(self):
        return self.discount_policy.apply(self.total)

order = Order(1000, PercentageDiscount(0.10))
print(order.final_total())
```

OUTPUT
Observed output:
900.0

Question	Prefer inheritance when...	Prefer composition when...
Relationship	The subtype is substitutable for the parent.	The object delegates work to a collaborator.
Change pattern	Shared behavior evolves together.	Policies or strategies change independently.
Risk	The hierarchy is shallow and stable.	A deep hierarchy would spread fragile assumptions.

8.3 Multiple inheritance and method resolution

Python supports multiple inheritance and resolves methods using a deterministic method resolution order. Mixins can be useful when they provide a narrow, stateless capability. Large multiple-inheritance graphs are difficult to reason about and should be treated as a design warning, not as a sign of sophistication.

Designing around responsibilities

Object-oriented design assigns responsibilities so a change has a clear home. A useful class protects an invariant, coordinates a coherent behavior, or represents a stable domain concept; turning every noun into a class usually creates ceremony rather than design.

Composition usually creates a clearer change boundary than inheritance. A RequestService can collaborate with a PolicyEvaluator, RequestRepository, and AuditRecorder without subclassing any of them, so each collaborator remains replaceable in tests. Use inheritance when the child genuinely satisfies the parent contract; Section 8.4 compares ABCs, Protocols, and informal duck typing. This keeps change local.

- Start from use cases and invariants, not from a class diagram.
- Keep domain rules separate from database, HTTP, and logging details.
- Use inheritance for substitutable behavior, not merely to reuse a few lines.
- Prefer constructor injection for required collaborators so dependencies are visible.

A boundary is useful when a change stops propagating. If changing one rule still forces edits through five layers, the interface added ceremony without reducing coupling.

Worked example: route a request through a replaceable policy

The policy is replaceable because the service depends on behavior, not on a concrete class. Change the rule and watch which files remain untouched.

PYTHON

```
from dataclasses import dataclass
from typing import Protocol

@dataclass(frozen=True)
class EquipmentRequest:
    employee_id: str
    item_type: str
    quantity: int

class ApprovalPolicy(Protocol):
    def requires_manager(self, request: EquipmentRequest) -> bool: ...

class StandardPolicy:
    def requires_manager(self, request: EquipmentRequest) -> bool:
        return request.quantity > 1 or request.item_type == "laptop"

class RequestService:
    def __init__(self, policy: ApprovalPolicy):
        self._policy = policy

    def route(self, request: EquipmentRequest) -> str:
        return "manager_review" if self._policy.requires_manager(request) else "approved"

service = RequestService(StandardPolicy())
print(service.route(EquipmentRequest("E-100", "laptop", 1)))
```

Expected result: manager_review

Try a second policy. If persistence code changes, the boundary is not as clean as it looks.

Trade-offs and failure modes

Multiple inheritance is not automatically wrong, but it demands cooperative design. Every participating class must use super() consistently and accept compatible arguments, because Python follows the method resolution order rather than calling a single obvious parent. Mixins should be narrow, stateless where possible, and named for the behavior they contribute. A large inheritance lattice with constructors that call specific parents directly is difficult to extend safely.

A design can also become too fragmented. Splitting a ten-line rule across six interfaces and factories does not create architecture; it creates navigation cost. Introduce an abstraction only when there is a real variation point, a testing seam, a policy boundary, or a dependency that should not leak into the domain. Simplicity is not the absence of structure. It is the minimum structure that keeps change predictable.

Data classes are appropriate for records and value objects, but they do not remove the need for invariants. frozen=True prevents normal field assignment but does not recursively freeze mutable values stored inside the object. Validation can live in __post_init__, a named constructor, or a separate parser depending on whether invalid values should ever exist in memory.

- A base class that exposes methods some children cannot support.
- Subclass checks scattered through callers, proving substitution has failed.
- Service classes that import database drivers directly into domain decisions.
- Interfaces with only one implementation and no testing or change boundary.
- Mixins that store shared mutable state or require fragile constructor ordering.
- God objects that own validation, persistence, notification, auditing, and presentation.

8.4 ABC, Protocol, and informal duck typing

An abstract base class creates nominal membership: the implementation explicitly inherits from or registers with the abstraction. A Protocol describes required structure for static analysis, allowing any compatible object to satisfy the interface. Informal duck typing relies on runtime behavior without a declared contract.

Choose according to the boundary. Use an ABC when runtime membership, shared implementation, or controlled construction matters. Use a Protocol when callers need a narrow behavioral contract and implementations should remain structurally independent.

Approach	Strength	Cost
Duck typing	Minimal ceremony	Failures appear only at runtime
Protocol	Static structural contract	Requires type-checker discipline
ABC	Runtime and nominal contract	Creates inheritance coupling

8.5 Refactoring a god object

A class that validates requests, evaluates policy, writes SQL, sends email, and formats HTTP responses is not cohesive. The problem is not its line count; it is that unrelated changes collide in one place. Separate domain policy, repository, notification, and transport responsibilities, then keep one application service to coordinate the use case.

The service should depend on narrow interfaces supplied through its constructor. Tests can then use an in-memory repository and a recording notifier while production wires PostgreSQL and email implementations.

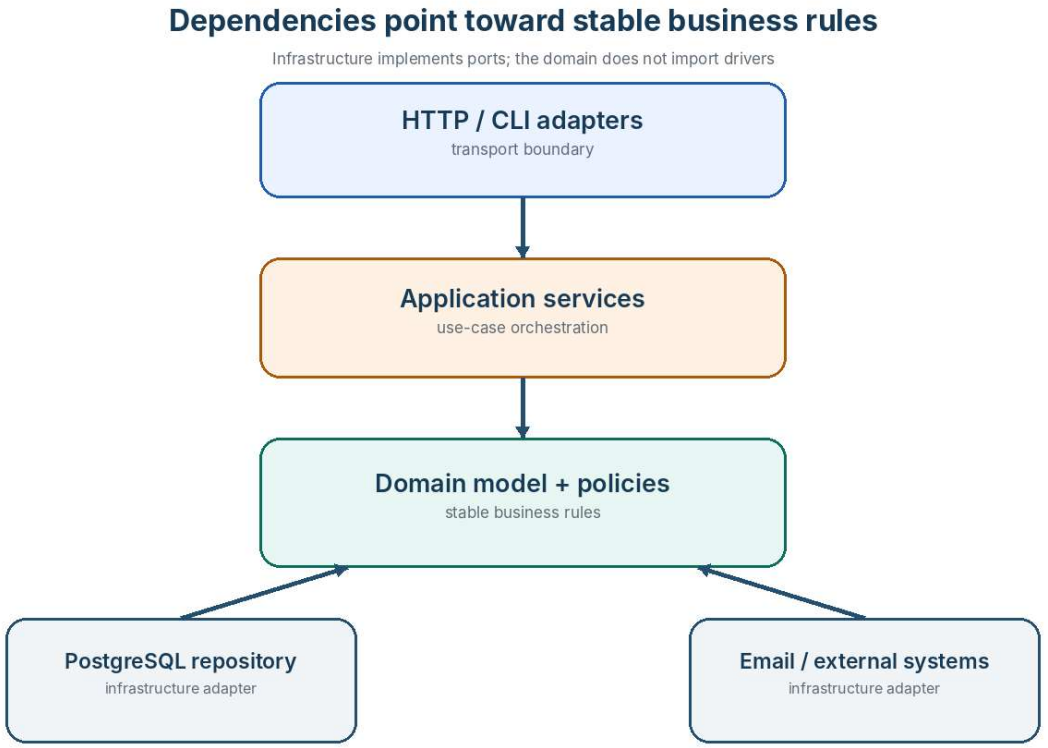

Figure 8.1 - Dependencies point toward stable business rules rather than outward to frameworks.

Build: split a god object without creating ceremony

Begin with the bad class. Name its responsibilities before extracting anything; otherwise refactoring becomes random file movement.

9. Take a single EquipmentManager class that validates, approves, saves, and emails requests. List its responsibilities before changing code.
10. Extract a pure approval policy and define the smallest useful Protocol for it.
11. Extract persistence behind a repository interface. Use an in-memory repository for tests.
12. Keep orchestration in a service that coordinates policy, repository, and audit recording.
13. Write tests proving that approval rules can change without altering persistence code.

What the refactor should prove

- Before-and-after responsibility map
- Pure policy tests
- Repository fake
- One rejected abstraction
- A dependency-direction check

Practice

14. Replace PercentageDiscount with a FixedDiscount strategy.
15. Add an SmsNotifier that satisfies the Notifier contract.
16. Describe a case where a mixin would be clearer than another level of inheritance.

Chapter 9 - Functions, Closures, and Decorators

Use functions as values without hiding state or control flow.

Learning objectives

- identify first-class and higher-order functions
- distinguish nested functions from closures
- write decorators that preserve metadata
- use caching responsibly

Approval rules increasingly need to be passed, wrapped, cached, or configured. Python's treatment of functions as objects makes those designs possible, but it also makes it easy to hide state and control flow.

9.1 First-class functions

Functions can be assigned to names, stored in collections, passed as arguments, and returned from other functions. A higher-order function accepts another function, returns one, or both. This is the foundation of map, filter, callbacks, strategy functions, and decorators.

9.2 A nested function is not automatically a closure

A closure is a function that retains access to one or more variables from an enclosing lexical scope. Merely defining a function inside another function is not enough; the inner function must capture an enclosing variable.

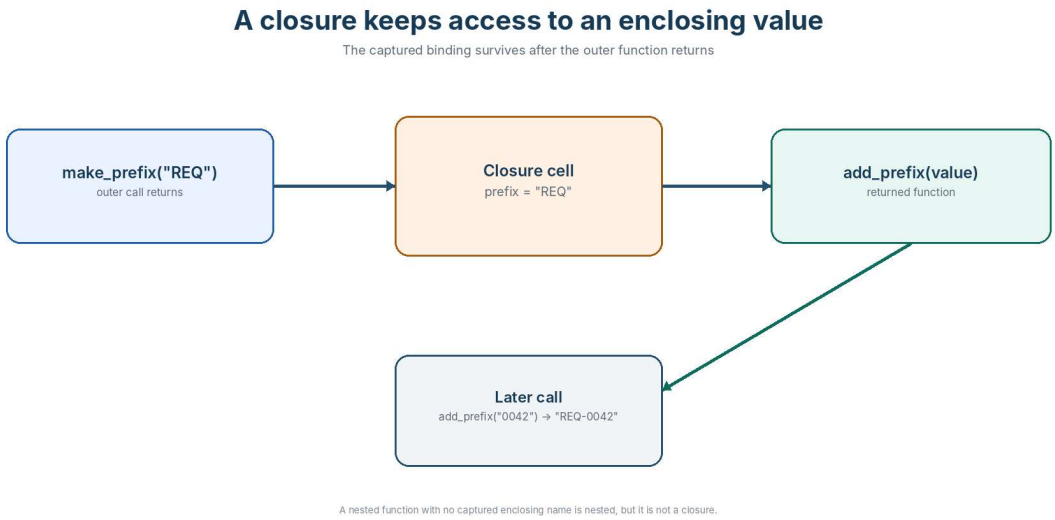

Figure 9.1 - A closure retains an enclosing value after the outer call returns.

Example 9.1 - A real closure captures an enclosing variable

```
PYTHON
def make_multiplier(factor):
    def multiply(value):
        return value * factor
    return multiply
```

```
triple = make_multiplier(3)
print(triple(7))
print(triple.__closure__ is not None)
```

OUTPUT
Observed output:
21
True

9.3 Decorators

A decorator replaces a function or class with the result of calling another callable. The @ syntax is convenient, but the underlying operation is ordinary assignment. Use functools.wraps so debugging, documentation, and introspection still see the original function metadata.

Example 9.2 - A decorator that preserves metadata

PYTHON
```
from functools import wraps
from time import perf_counter

def timed(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        started = perf_counter()
        result = func(*args, **kwargs)
        elapsed = perf_counter() - started
        print(f"{func.__name__} completed in {elapsed:.6f}s")
        return result
    return wrapper

@timed
def add(a, b):
    """Return the sum."""
    return a + b

print(add(2, 5))
print(add.__name__)
```

OUTPUT
Observed output:
add completed in 0.000001s
7
add

9.4 Caching and invalidation

Caching trades memory and staleness risk for speed. lru_cache works best for deterministic functions with hashable arguments. It is a poor fit when results depend on mutable external state unless you define an explicit invalidation policy.

Example 9.3 - Memoization with lru_cache

PYTHON
```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(20))
print(fibonacci.cache_info().hits > 0)
```

OUTPUT
Observed output:
6765

True

Design rule: preserve state and function contracts

Use closures for small captured state and classes when state grows into several operations, invariants, or lifecycle concerns. Decorators should stay focused on cross-cutting behavior, preserve metadata with functools.wraps, and match the wrapped execution model: when decorator behavior must surround async execution, use an async wrapper and await the wrapped coroutine. A synchronous wrapper may return the coroutine unchanged, but it cannot observe its awaited completion.

Worked example: preserve behavior while adding timing

The wrapper adds timing but must not erase the original function's name, documentation, return value, or exception behavior.

PYTHON

```
from functools import wraps
from time import perf_counter

def count_calls(func):
    calls = 0

    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal calls
        calls += 1
        started = perf_counter()
        try:
            return func(*args, **kwargs)
        finally:
            elapsed = perf_counter() - started
            print(f"{func.__name__}: call={calls}, elapsed={elapsed:.6f}s")

    return wrapper

@count_calls
def add(a: int, b: int) -> int:
    return a + b

print(add(2, 3))
print(add(10, 5))
```

Expected result: a timing line, then 5; a second timing line, then 15. Exact elapsed values vary.

Now break it with an async function. The failure explains why synchronous and asynchronous decorators need different wrappers.

Trade-offs and failure modes

Caching and retries are behavior contracts, not free optimizations. Cache only when the same arguments may safely reuse a result, define invalidation for changing data, and retry side effects only when idempotency or transactional protection prevents duplication. A wrapper must preserve return values, exceptions, metadata, and sync/async semantics.

- Using a synchronous wrapper when the decorator must observe or handle completion of an async function.
- Forgetting functools.wraps and breaking introspection or framework registration.
- Caching functions whose results depend on changing external state.
- Retrying non-idempotent actions without deduplication or transactional protection.

Build: add behavior without erasing a function contract

Implement the closure and decorator, then write tests that protect metadata, return values, and exception identity.

17. Build a function factory named make_validator(minimum, maximum) that returns a closure validating numeric inputs.
18. Use nonlocal to count successful validations and expose the count through a small helper or attribute.
19. Create a decorator that logs the function name and re-raises failures without changing their type.
20. Add functools.wraps and verify that the decorated function retains __name__ and __doc__.
21. Write a paragraph explaining whether the design should remain a closure or become a class as requirements grow.

Contract checks

- Metadata-preservation test
- Expected exception test
- Async incompatibility demonstration
- Cache invalidation note
- Final decorator contract

Practice

22. Write a closure that generates request identifiers with a prefix.
23. Create a decorator that rejects negative numeric arguments.
24. Add cache_clear() to a scenario where the underlying data changes.

Chapter 10 - Iteration, Generators, and Context Managers

Process values lazily and make cleanup explicit.

Learning objectives

- distinguish iterables and iterators
- build a generator
- explain lazy evaluation
- write safe context managers

The request workflow begins processing collections that may be large or tied to files and connections. Iterators and generators let work happen one value at a time; context managers make ownership and cleanup explicit.

10.1 Iterable versus iterator

An iterable can produce an iterator, usually through __iter__. An iterator represents one traversal and implements __next__. Calling iter() on a list creates a fresh iterator; calling iter() on an iterator normally returns the same iterator.

Example 10.1 - A custom iterator

PYTHON

```
class Countdown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < 0:
            raise StopIteration
        value = self.current
        self.current -= 1
        return value

print(list(Countdown(3)))
```

OUTPUT

```
Observed output:
[3, 2, 1, 0]
```

10.2 Generators

A generator function contains yield. Each yield returns a value and suspends execution while preserving local state. Generators are ideal when the full result need not exist in memory at once. They do not guarantee that only one value exists anywhere in memory; they simply avoid materializing the entire produced sequence by default.

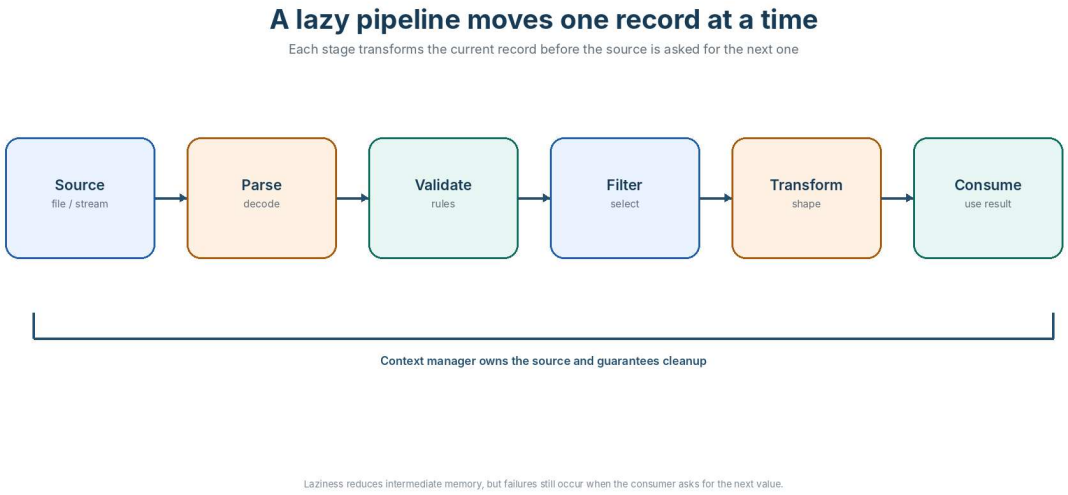

Figure 10.1 - A lazy pipeline moves one record through each stage before requesting the next.

Example 10.2 - Generator-based streaming

```
PYTHON
def even_numbers(limit):
    for value in range(limit):
        if value % 2 == 0:
            yield value

print(list(even_numbers(8)))
```

```
OUTPUT
Observed output:
[0, 2, 4, 6]
```

10.3 Context managers

A context manager defines a controlled entry and exit around a block. The with statement guarantees that exit logic runs whether the block succeeds or raises an exception. Returning True from __exit__ suppresses the exception; do this only when suppression is intentional and documented.

Example 10.3 - A safe custom context manager

```
PYTHON
from contextlib import contextmanager

@contextmanager
def managed_resource(name):
    print(f"acquire {name}")
    try:
        yield name
    finally:
        print(f"release {name}")

with managed_resource("connection") as resource:
    print(f"using {resource}")
```

```
OUTPUT
Observed output:
acquire connection
using connection
release connection
```

Building lazy processing pipelines

Iteration separates the source of values from the consumer of values. An iterable can produce an iterator; the iterator owns traversal state and yields one item at a time through __next__. Many iterators are one-shot. Once exhausted, calling iter on the same iterator returns the iterator itself, not a fresh traversal. This is why code that consumes a generator to calculate a count cannot then loop over it again without recreating it.

Generators make stateful iteration concise. Every yield pauses execution while retaining local variables and the current instruction position. Generator expressions provide the same lazy behavior for simple transformations. Chaining small generators can process large files with stable memory usage because each stage handles only the current item rather than constructing full intermediate lists.

- Document whether a function returns a reusable collection or a one-shot iterator.
- Keep generator stages pure where possible so failures are easier to locate.
- Close files, sockets, locks, and transactions with context managers, not with hope.

Generators move work - and failures - to consumption time. Test the iteration path, including early exit and cleanup, not just the line that creates the generator.

Worked example: stream records and guarantee cleanup

The pipeline has two contracts: values arrive lazily, and the resource closes whether consumption succeeds or fails.

```
PYTHON
from contextlib import contextmanager
from io import StringIO

def non_empty_lines(stream):
    for raw_line in stream:
        line = raw_line.strip()
        if line and not line.startswith("#"):
            yield line

@contextmanager
def sample_stream():
    stream = StringIO("# header\n laptop \n\nmonitor\n")
    try:
        yield stream
    finally:
        stream.close()

with sample_stream() as source:
    print(list(non_empty_lines(source)))
```

Expected result: ['laptop', 'monitor']

Consume the generator twice. The second result makes the one-shot contract impossible to ignore.

Trade-offs and failure modes

Lazy evaluation moves both work and failure to consumption time. Test the iteration path, not merely generator creation. If a caller needs immediate validation or repeated access, returning a concrete collection can be the clearer contract. Keep generator stages simple and explicit.

Context managers define setup and teardown around a lexical block. __exit__ receives exception information and may suppress the exception by returning a truthy value. Suppression should be deliberate and narrow; returning True accidentally can erase production failures. The contextlib.contextmanager decorator is concise, but the code before yield is setup and the code in finally after yield is teardown. Omitting finally can skip cleanup when the block raises.

Asynchronous iterators and asynchronous context managers apply the same ideas to non-blocking resources. They use __aiter__, __anext__, __aenter__, and __aexit__ with async for and async with. Do not put blocking file or network operations into an async iterator and assume syntax makes them non-blocking.

- Attempting to iterate a consumed generator a second time.
- Returning a generator where the caller expects validation to happen immediately.
- Catching StopIteration manually inside ordinary iteration logic.
- Forgetting finally in a generator-based context manager.
- Suppressing exceptions unintentionally from __exit__.
- Holding a database transaction open while yielding a long stream to a slow client.

10.4 Failure timing in lazy code

A generator function returns a generator object before its body performs normal work. Validation placed before the first yield still runs only when iteration begins. This shifts failure from construction to consumption and can surprise callers that expect immediate validation.

Document the timing. Validate eagerly before returning the generator when the input contract must fail immediately, or make the lazy failure explicit in the API and tests.

```
PYTHON
def records(source):
    if source is None:
        raise ValueError("source is required")
    for line in source:
        yield line.strip()

generator = records(None)
# The ValueError occurs on next(generator), not above.
```

10.5 ExitStack for a dynamic resource set

A normal with statement is ideal when the number of resources is fixed in source code. ExitStack handles a resource set discovered at runtime. Every successfully entered context is registered for reverse-order cleanup, including when a later acquisition fails.

This pattern is useful for a variable collection of files, locks, temporary directories, or optional transactions. It is not a reason to hold resources longer than necessary.

```
PYTHON
from contextlib import ExitStack
from pathlib import Path

paths = [Path("a.txt"), Path("b.txt")]
with ExitStack() as stack:
    streams = [stack.enter_context(path.open("w", encoding="utf-8")) for path in paths]
    for stream in streams:
        stream.write("ok\n")
```

Build: process a large request stream safely

Use enough records to make laziness meaningful. Tests must consume the generator, because creation alone proves almost nothing.

25. Create a generator that reads request records from an iterable of text lines and yields validated dictionaries.
26. Add a second generator that filters requests by status without building an intermediate list.
27. Use itertools.islice to inspect the first five results while leaving the rest lazy.
28. Wrap the source in a context manager and prove through a test that cleanup occurs after both success and failure.
29. Measure peak memory for a list-based pipeline and a generator pipeline using a realistic number of records.

Streaming evidence

- Peak-memory result
- Cleanup-on-failure test
- One-shot iterator demonstration
- Documented validation timing
- Final pipeline

Practice

30. Create a generator that yields lines from a file after stripping whitespace.
31. Make Countdown reject negative starting values.
32. Modify the context manager to record whether the block raised an exception.

PART III - RELIABILITY AND CONCURRENCY

Measure behavior, preserve failures, and choose the execution model that matches the workload.

Chapter 11 - Data Structures, Algorithms, and Complexity

Choose structures and algorithms from the operations the program performs.

Learning objectives

- select specialized containers
- reason about average and worst-case complexity
- avoid false O(1) claims
- implement a shortest-path algorithm

As the number of requests grows, data-structure choices stop being cosmetic. The operation performed most often - lookup, membership, ordering, queueing, or path finding - should drive the representation.

11.1 Specialized containers

The collections and heapq modules cover common needs that are awkward with plain lists and dictionaries. Counter counts hashable values. defaultdict supplies a value for missing keys. deque provides efficient operations at both ends. heapq maintains a min-heap for priority work.

Example 11.1 - Counter, defaultdict, deque, and heapq

PYTHON

```
import heapq
from collections import Counter, defaultdict, deque

words = "red blue red green red blue".split()
counts = Counter(words)

grouped = defaultdict(list)
for word in words:
    grouped[word[0]].append(word)

queue = deque(["first", "second"])
queue.appendleft("urgent")

priorities = []
heapq.heappush(priorities, (2, "normal"))
heapq.heappush(priorities, (1, "high"))

print(counts.most_common(2))
print(dict(grouped))
print(queue.popleft())
print(heapq.heappop(priorities))
```

OUTPUT

```
Observed output:
[('red', 3), ('blue', 2)]
{'r': ['red', 'red', 'red'], 'b': ['blue', 'blue'], 'g': ['green']}
urgent
(1, 'high')
```

11.2 Complexity is tied to the operation

Operation	Typical complexity	Reason
list or tuple membership	O(n)	Elements may need to be scanned

Operation	Typical complexity	Reason
		sequentially.
set or dict lookup	Average O(1), worst O(n)	Hashing usually jumps to a bucket; collisions can degrade behavior.
deque append/popleft	O(1)	The structure is optimized for both ends.
heap push/pop	O(log n)	The heap property is restored along a tree path.
sorting	O(n log n)	Python uses Timsort with strong real-world behavior.

Correction: Tuple membership is not constant time. A literal tuple may be compiled efficiently, but membership still requires comparison against elements until a match is found or the tuple is exhausted. Set membership is average O(1), not an unconditional guarantee.

Use Big O to rule out designs that scale badly; use measurements to decide whether the real workload needs optimization.

Example 11.2 - Set and tuple membership have different complexity

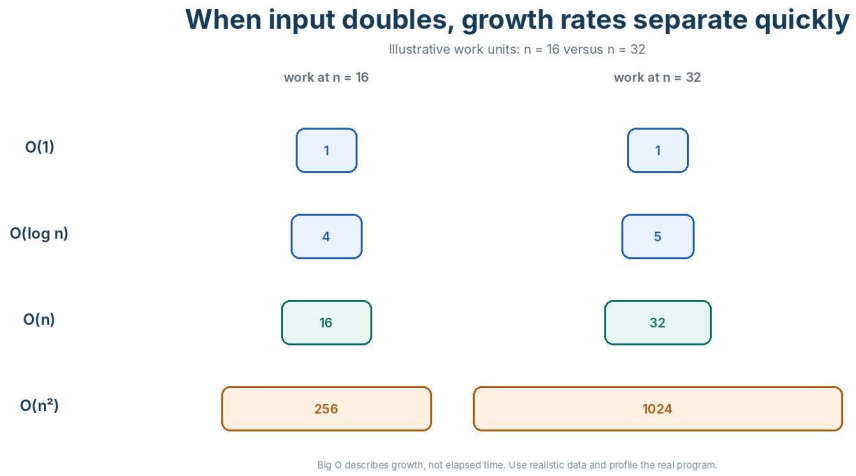

Figure 11.1 - When input doubles, constant, logarithmic, linear, and quadratic work grow differently.

PYTHON
```
values_list = list(range(10_000))
values_tuple = tuple(values_list)
values_set = set(values_list)

target = 9_999
print(target in values_tuple)
print(target in values_set)
```

OUTPUT
Observed output:
True
True

11.3 Graphs and shortest paths

Dijkstra's algorithm computes shortest paths from one source when edge weights are non-negative. A priority queue ensures that the next most promising node is processed efficiently. The algorithm is not valid for graphs with negative edge weights.

Example 11.3 - Dijkstra shortest paths

PYTHON

```
import heapq

def dijkstra(graph, start):
    if start not in graph:
        raise ValueError("Start node must appear as a graph key")
    for edges in graph.values():
        for neighbor, weight in edges:
            if neighbor not in graph:
                raise ValueError("Every neighbor must appear as a graph key")
            if weight < 0:
                raise ValueError("Dijkstra requires non-negative edge weights")

    distances = {node: float("inf") for node in graph}
    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_distance, current = heapq.heappop(queue)
        if current_distance != distances[current]:
            continue
        for neighbor, weight in graph[current]:
            candidate = current_distance + weight
            if candidate < distances[neighbor]:
                distances[neighbor] = candidate
                heapq.heappush(queue, (candidate, neighbor))
    return distances

graph = {
    "A": [("B", 1), ("C", 4)],
    "B": [("C", 2), ("D", 5)],
    "C": [("D", 1)],
    "D": [],
}
print(dijkstra(graph, "A"))
```

OUTPUT

Observed output:

```
{'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Worked example: shortest paths with stated assumptions

The code is less important than its assumptions: graph direction, every referenced neighbor appearing as a graph key, non-negative weights, and what “shortest” means.

PYTHON

```
import heapq

def shortest_paths(graph, start):
    if start not in graph:
        raise ValueError("Start node must appear as a graph key")
    for edges in graph.values():
        for neighbor, weight in edges:
            if neighbor not in graph:
                raise ValueError("Every neighbor must appear as a graph key")
            if weight < 0:
                raise ValueError("Dijkstra requires non-negative edge weights")

    distances = {node: float("inf") for node in graph}
    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_distance, node = heapq.heappop(queue)
        if current_distance != distances[node]:
            continue
        for neighbor, weight in graph[node]:
            candidate = current_distance + weight
            if candidate < distances[neighbor]:
                distances[neighbor] = candidate
                heapq.heappush(queue, (candidate, neighbor))
    return distances

graph = {
```

```
    "A": [("B", 2), ("C", 5)],
    "B": [("C", 1)],
    "C": [],
}
print(shortest_paths(graph, "A"))
```

Expected result: {'A': 0, 'B': 2, 'C': 3}

Add a negative edge. The validation above raises ValueError before Dijkstra can return a misleading result.

Trade-offs and failure modes

Complexity predicts growth, not elapsed time. Keep the assumptions attached to the operation: list and tuple membership are linear, set and dictionary lookup are average constant time, heaps expose priority rather than global sort order, and Dijkstra requires non-negative weights. Use realistic inputs and validate graph structure before trusting algorithm output.

- Using list.pop(0) repeatedly for a high-volume queue instead of deque.
- Assuming a heap's internal list is fully sorted.
- Using mutable or hash-unstable objects as dictionary keys or set members.
- Running Dijkstra on negative edge weights or benchmarking unrealistic inputs.

11.4 BFS, Dijkstra, and the weight question

Breadth-first search finds a shortest path by edge count in an unweighted graph. Dijkstra finds minimum total cost when weights are non-negative. Treating every graph problem as Dijkstra adds unnecessary machinery; treating weighted costs as unweighted produces the wrong answer.

Before choosing an algorithm, define what a node, edge, direction, and weight mean. A routing graph may weight approval steps by expected delay, monetary cost, or risk. Those are different models and can produce different paths.

- Unweighted shortest path: start with BFS.
- Non-negative weighted path: Dijkstra.
- Negative weights: use an algorithm designed for them.
- All-pairs paths on small graphs: consider Floyd-Warshall.

11.5 Reproducible benchmarking

A benchmark must state interpreter, hardware, input generation, warm-up, repetitions, and the statistic reported. A single timing is evidence of one run, not a general performance claim. Use timeit for small isolated operations and a profiler for application behavior.

Benchmark realistic sizes and distributions. Set membership often wins dramatically for repeated lookup, but converting a one-use list to a set can cost more than the saved lookup. Measure the complete operation.

A benchmark without inputs, environment, repetitions, and measurement method is an anecdote. Record the method beside the result.

Build: choose structures, then prove the choice

Write the operations before selecting a container. Measure the complete workflow, not one flattering micro-operation.

33. Implement a request queue first with a list and then with deque. Benchmark removal from the left for increasing sizes.
34. Use Counter to summarize equipment types and most_common to report the top three.

35. Create a priority queue that stores priority, insertion sequence, and request ID so equal priorities remain safe.
36. Model an approval-routing graph and compare BFS with Dijkstra for unweighted and weighted routes.
37. Write a complexity table for every operation used in the lab and confirm it against measured behavior.

Benchmark record

- Operation list
- Complexity prediction
- Reproducible benchmark method
- Measured result
- Explanation of any mismatch

Practice

38. Use deque to implement a fixed-length activity log.
39. Benchmark tuple and set membership with timeit.
40. Modify Dijkstra so that it returns predecessor links and reconstructs a path.

Chapter 12 - Errors, Logging, Testing, and Profiling

Make failures diagnosable, behavior testable, and performance measurable.

Learning objectives

- design custom exceptions
- preserve failure causes
- write focused tests
- profile before optimizing

Once the service has real inputs and dependencies, failures need to become observable evidence. Exceptions, logs, tests, and profiles answer different questions and should not be collapsed into one reliability mechanism.

12.1 Exceptions are part of the interface

Catch an exception when you can add context, recover, translate it into a domain error, or guarantee cleanup. Catching Exception merely to print and continue usually hides failure. Exception chaining with raise ... from ... preserves the original cause.

Example 12.1 - Custom exceptions and exception chaining

PYTHON

```
class ConfigurationError(Exception):
    pass

def parse_port(raw_value):
    try:
        port = int(raw_value)
    except ValueError as exc:
        raise ConfigurationError("port must be an integer") from exc
    if not 1 <= port <= 65535:
        raise ConfigurationError("port is outside the valid range")
    return port

try:
    parse_port("not-a-number")
except ConfigurationError as exc:
    print(type(exc.__cause__).__name__)
    print(exc)
```

OUTPUT

```
Observed output:
ValueError
port must be an integer
```

12.2 Logging with context

Logs should answer what happened, where, and to which request or business entity. Use parameterized logging instead of building strings eagerly. Avoid logging passwords, tokens, personal data, or complete request bodies without a documented reason.

During an incident, “failed to process request” is nearly useless. Add request and domain identifiers while the system is healthy, before several failures overlap.

Example 12.2 - Structured logging

PYTHON

```
import logging

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")
logger = logging.getLogger("book")

request_id = "req-123"
logger.info("request completed request_id=%s status=%s", request_id, "approved")
```

OUTPUT
Observed output:
INFO request completed request_id=req-123 status=approved

12.3 Testing behavior

A test should describe observable behavior and isolate the cause of failure. Pure functions are easy to test because their output depends only on input. Integration tests verify boundaries such as databases, files, and HTTP services. End-to-end tests verify critical user flows, but they are slower and more fragile.

Example 12.3 - Unit testing a pure function

PYTHON
```
import unittest

def calculate_total(items):
    return sum(items)

class TotalTests(unittest.TestCase):
    def test_calculate_total(self):
        self.assertEqual(calculate_total([10, 20, 30]), 60)

suite = unittest.defaultTestLoader.loadTestsFromTestCase(TotalTests)
result = unittest.TextTestRunner(verbosity=0).run(suite)
print(result.wasSuccessful())
```

OUTPUT
Observed output:
--
Ran 1 test in 0.000s

OK
True

12.4 Profiling and optimization

Do not optimize code because it looks slow. Measure representative workloads with timeit, cProfile, or an application profiler. Improve the dominant bottleneck, then measure again. Replacing clear code with clever code without evidence is not optimization; it is speculative complexity.

Example 12.4 - Measure before optimizing

PYTHON
```
import timeit

list_time = timeit.timeit("sum([i * i for i in range(1000)])", number=100)
generator_time = timeit.timeit("sum(i * i for i in range(1000))", number=100)

print(list_time > 0)
print(generator_time > 0)
```

OUTPUT
Observed output:
True
True

Making failure observable and testable

Exceptions are part of an application's public behavior. A caller needs to know which failures are expected, which can be retried, and which indicate a programming defect. Domain-specific exceptions should express meaningful conditions such as InsufficientBalance or InvalidTransition. They should not merely rename built-in exceptions without improving the contract. Low-level exceptions can be translated at a boundary while preserving the original cause with raise NewError(...) from exc.

Logging is evidence, not decoration. A useful event states what happened, where it happened, and which request or entity was involved. Structured fields are easier to search than prose assembled through string concatenation. Correlation IDs connect events across HTTP handlers, services, repositories, and external calls, but they do not replace stable domain identifiers such as request_id or employee_id. Each identifier answers a different question.

Tests should be organized around behavior and risk. Unit tests cover pure rules and small collaborators. Integration tests verify boundaries such as SQL queries, serialization, and framework routing. End-to-end tests cover a few critical workflows through the deployed interface. A large suite of slow end-to-end tests is not automatically stronger than a balanced test pyramid with clear ownership of each failure mode.

- Catch an exception only where the code can add context, recover, translate, or clean up.
- Log once at the boundary that owns the failure response; avoid duplicating the same traceback at every layer.
- Use deterministic fixtures and isolate external services behind interfaces.
- Profile before optimizing and compare the result after every change.

Coverage can rise while confidence stays flat. One precise test of a business failure often protects more behavior than several assertions against implementation details.

Worked example: make a state transition observable

The transition either succeeds visibly or fails with a named reason. Logging records the outcome; it does not replace the exception contract.

PYTHON

```
import logging
from dataclasses import dataclass

logging.basicConfig(level=logging.INFO, format="%(levelname)s %(message)s")
logger = logging.getLogger(__name__)

class InvalidTransition(Exception):
    pass

@dataclass
class Request:
    request_id: str
    status: str = "submitted"

    def approve(self) -> None:
        if self.status != "submitted":
            raise InvalidTransition(f"cannot approve from {self.status}")
        self.status = "approved"

request = Request("REQ-1")
request.approve()
logger.info("request approved request_id=%s status=%s", request.request_id, request.status)
print(request.status)
```

Expected result: An INFO log is emitted and approved is printed.

Force an invalid transition and inspect both the exception and the log. They serve different readers.

Trade-offs and failure modes

Broad exception handling at process boundaries can be necessary to return a controlled error or keep a worker alive, but the handler must record the traceback and preserve a safe response. Broad handling deep inside business logic often converts distinct failures into the same None or False value, making diagnosis and recovery impossible. Catch specific exceptions when behavior differs.

Mocking can isolate code, but excessive mocks lock tests to implementation details. Prefer fakes for stateful boundaries such as repositories, and use real value objects and policies. A test that asserts the exact sequence of private method calls may fail during harmless refactoring while missing the behavior users rely on. Assert outputs, state transitions, emitted events, and boundary interactions that are part of the contract.

Profiling tools answer different questions. timeit compares small code paths under repeated execution. cProfile shows function-level CPU time for a program. Sampling profilers reduce observer overhead in long-running systems. Memory tools find growth and allocation pressure. Optimize the measured bottleneck, then rerun correctness tests because faster incorrect code remains incorrect.

- except Exception: pass or returning a default value after an unexpected failure.
- Logging secrets, access tokens, passwords, or full personal records.
- Creating a new correlation ID in every layer instead of propagating one request context.
- Writing tests that depend on execution order or real network availability.
- Mocking the method under test rather than its external boundary.
- Optimizing code without a baseline profile or regression measurement.

12.5 Mapping domain failures to boundaries

Domain code should raise errors meaningful to the use case, while the HTTP or command-line adapter translates them into its own response language. The domain should not raise HTTPException because it may also be used from a worker, test, or command-line tool.

Preserve the distinction between invalid input, missing data, conflict, temporary infrastructure failure, and unexpected defects. That distinction drives status codes, retry policy, log level, and alerting.

Condition	Domain error	Typical HTTP result
Invalid field	ValidationError	422 or 400
Missing request	NotFound	404
Invalid state transition	Conflict	409
Temporary database failure	TemporaryFailure	503
Unexpected defect	Uncaught / internal	500

12.6 A balanced testing portfolio

Unit tests give fast feedback on rules. Integration tests verify SQL, serialization, framework routing, and other boundaries. Contract tests protect assumptions between independently changing services. End-to-end tests prove a few critical flows through the deployed interface.

The pyramid is not a quota. The right distribution follows risk and feedback cost. A data-access library may need more integration tests; a policy engine may need many unit properties.

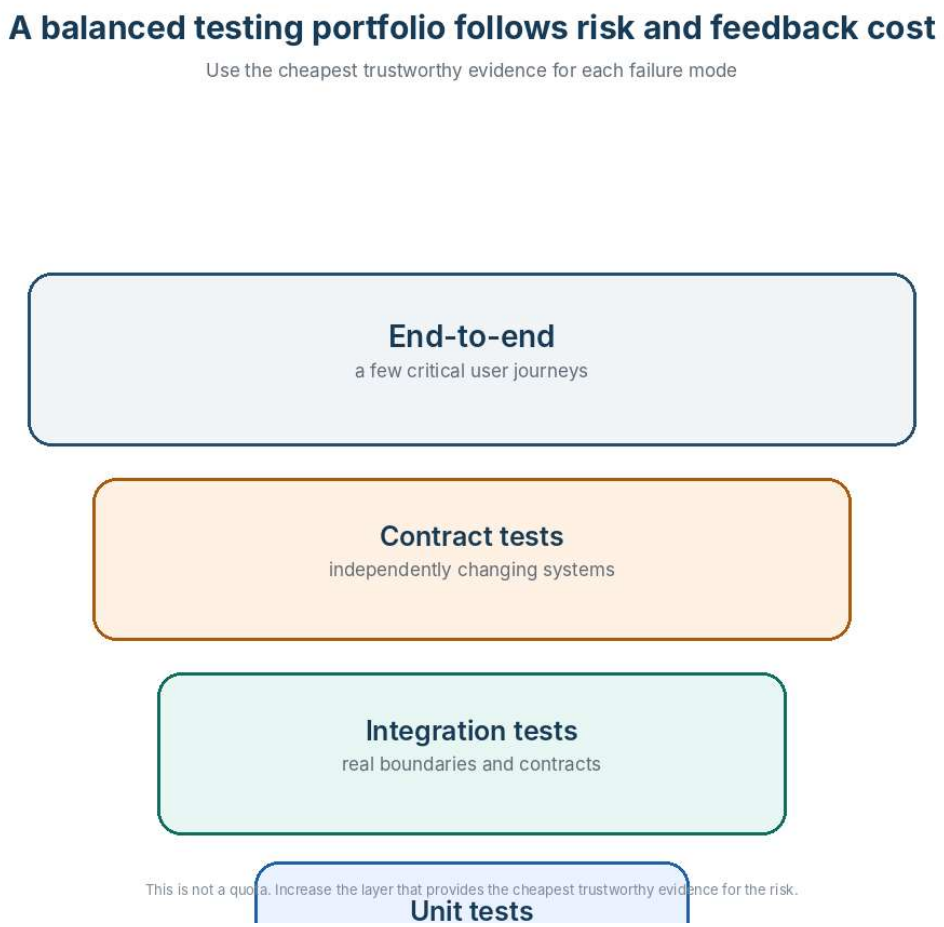

Figure 12.1 - A balanced testing portfolio favors fast feedback without abandoning system-level proof.

Build: turn one vague failure into useful evidence

Start from the user report "approval failed." Build the exception, log fields, test, and profile that would let another engineer diagnose it.

41. Define a small exception hierarchy for invalid input, missing data, invalid state transition, and temporary infrastructure failure.
42. Translate a simulated database KeyError into a domain NotFound error using exception chaining.
43. Add structured log fields for correlation_id and request_id without logging sensitive request details.
44. Write unit tests for transition rules and an integration-style test using an in-memory repository fake.
45. Profile a deliberately slow report function, optimize the measured hot path, and record before-and-after timings.

Diagnostic evidence

- Exception map
- Redacted structured logs
- Unit and boundary tests
- Before-and-after profile
- A diagnosis another engineer can repeat

Key takeaways

- Exceptions: Translate failures at boundaries and preserve causes.
- Logging: Add identifiers and outcomes without exposing secrets.
- Tests: Verify behavior at the right level.
- Profiling: Measure, change one bottleneck, measure again.

Practice

46. Add a test for an empty list to Example 12.3.
47. Create a domain exception for an invalid request status transition.
48. Use cProfile on a function that sorts and aggregates a large list.

Chapter 13 - Threads and Processes

Match threads and processes to the work they can actually improve.

Learning objectives

- distinguish concurrency and parallelism
- choose threads for waiting work
- choose processes for CPU work
- avoid unsafe shared state

The service now spends time waiting on I/O and may also perform CPU-heavy work. Threads and processes solve different bottlenecks, and using the wrong one can add complexity without increasing throughput.

13.1 Concurrency versus parallelism

Concurrency means multiple tasks make progress during overlapping periods. Parallelism means work executes at the same physical time. In the standard GIL-enabled CPython 3.13 build used for this book, the global interpreter lock limits simultaneous execution of Python bytecode by multiple threads, but threads remain effective when tasks spend time waiting for I/O. Python 3.13 also offers an optional experimental free-threaded build; in Python 3.14, free-threaded Python remains optional but is officially supported rather than experimental. Check the actual interpreter build rather than inferring GIL behavior from the version number. [PY-THREAD-313] [PY-FREE-314]

> **Note: Version caution: Free-threaded CPython builds are an evolving option. Do not assume a deployment is free-threaded merely because the Python version is recent. Check the actual interpreter build and verify third-party extension compatibility.**

13.2 Threads for waiting work

Thread pools are convenient for blocking network, file, or database calls. Shared memory is easy to access and therefore easy to corrupt. Prefer immutable data, queues, and narrow synchronization boundaries.

Example 13.1 - ThreadPoolExecutor for I/O-style waiting

PYTHON

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def fetch(name):
    sleep(0.05)
    return f"finished {name}"

with ThreadPoolExecutor(max_workers=3) as executor:
    results = list(executor.map(fetch, ["A", "B", "C"]))

print(results)
```

OUTPUT

```
Observed output:
['finished A', 'finished B', 'finished C']
```

13.3 Processes for CPU-bound work

Processes have separate interpreters and memory spaces. They can use multiple CPU cores for pure Python CPU work, but process startup and data serialization add overhead. When using multiprocessing or ProcessPoolExecutor, protect executable process setup with if __name__ == "__main__" and keep worker functions importable at module scope; this is essential with spawn/forkserver start methods, not only on Windows.

Example 13.2 - ProcessPoolExecutor for CPU-bound work

PYTHON

```
from concurrent.futures import ProcessPoolExecutor

def square(value):
    return value * value

if __name__ == "__main__":
    with ProcessPoolExecutor(max_workers=2) as executor:
        print(list(executor.map(square, [2, 3, 4])))
```

OUTPUT

Observed output:

```
[4, 9, 16]
```

Workload	Recommended starting point	Primary risk
Blocking I/O with a synchronous library	Thread pool	Race conditions and excessive threads
CPU-heavy pure Python calculation	Process pool	Serialization and startup overhead
Large number of async-native network operations	asyncio	Blocking calls inside the event loop

Worker count is a capacity decision, not a dial that always moves right. More workers can add scheduling overhead or saturate the database, API, or memory.

Worked example: bound a thread pool

The pool is bounded. That one constraint matters more than creating threads with impressive-looking syntax.

PYTHON

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def fetch(item_id: int) -> str:
    sleep(0.02)
    return f"item-{item_id}"

with ThreadPoolExecutor(max_workers=3) as executor:
    results = list(executor.map(fetch, range(5)))

print(results)
```

Expected result: ['item-0', 'item-1', 'item-2', 'item-3', 'item-4']

Raise inside one worker and call result(). Unobserved worker exceptions are not a resilience strategy.

Trade-offs and failure modes

Concurrency failures are usually ownership failures. Protect the whole invariant rather than one assignment, keep critical sections short, observe Future results so worker exceptions surface, and avoid nested waits that can exhaust a constrained pool. Free-threaded builds may change CPU

parallelism, but they do not remove races, synchronization needs, extension compatibility concerns, or measurement. [PY-FREE-313] [PY-FREE-314]

- Creating one thread or process per item instead of using a bounded pool.
- Assuming compound updates to shared state are automatically safe.
- Ignoring Future results and losing worker exceptions.
- Sending unpicklable work to a process pool or forgetting the main guard on spawn-based systems.

13.4 Protecting a compound invariant

A race condition occurs when correctness depends on an interleaving that is not controlled. Checking available stock and then decrementing it are two operations. Locking only the assignment allows two threads to pass the check and oversubscribe stock.

Protect the entire invariant or move ownership to one worker receiving commands through a queue. The second design often reduces shared-state reasoning at the cost of message coordination.

```python
from threading import Lock

lock = Lock()
available = 1

def reserve() -> bool:
    global available
    with lock:
        if available == 0:
            return False
        available -= 1
        return True
```

Build: find the point where concurrency begins to help

Use the same workload sequentially, with threads, and with processes. Record overhead as carefully as speedup.

49. Build a sequential simulation of five independent I/O waits and record elapsed time.
50. Run the same work with a ThreadPoolExecutor and compare throughput while keeping the pool bounded.
51. Introduce one worker failure and prove that inspecting its Future exposes the exception.
52. Implement a CPU-heavy pure function and compare sequential execution with ProcessPoolExecutor for several input sizes.
53. Document the crossover point where process overhead begins to pay for itself on your machine.

Concurrency record

- Workload and machine details
- Sequential baseline
- Thread/process result
- Observed exception propagation
- Crossover-point explanation

Practice

54. Change Example 13.1 to collect results as each future completes.
55. Measure when process overhead outweighs a small calculation.
56. Explain why adding more workers can reduce performance.

Chapter 14 - Asynchronous Programming with asyncio

Use cooperative concurrency with explicit task lifetime and cancellation.

Learning objectives

- create and await coroutines
- run tasks concurrently
- handle timeouts and cancellation
- propagate request context correctly

HTTP calls, database work, and notifications often spend most of their time waiting. asyncio can overlap that waiting efficiently, provided every dependency is actually non-blocking and every task has a defined lifetime.

14.1 Coroutines and the event loop

Calling an async def function creates a coroutine object. It does not run until it is awaited or scheduled. While a coroutine awaits an async operation, the event loop can run another ready task. A blocking call inside the event loop blocks every task sharing that loop.

Changing def to async def does not make a blocking client cooperative. While that call owns the event-loop thread, every other task on the loop waits.

Example 14.1 - Run independent coroutines concurrently

PYTHON

```
import asyncio

async def fetch(name, delay):
    await asyncio.sleep(delay)
    return f"{name} complete"

async def main():
    results = await asyncio.gather(
        fetch("A", 0.02),
        fetch("B", 0.01),
    )
    print(results)

asyncio.run(main())
```

OUTPUT

```
Observed output:
['A complete', 'B complete']
```

14.2 Timeouts, cancellation, and cleanup

Production async code needs explicit limits. Network calls can hang, downstream systems can slow down, and users can disconnect. Apply timeouts, handle cancellation, and release resources in finally blocks or async context managers.

Example 14.2 - Timeouts are part of async design

PYTHON

```
import asyncio

async def slow_operation():
    await asyncio.sleep(0.1)
```

```
async def main():
    try:
        await asyncio.wait_for(slow_operation(), timeout=0.01)
    except TimeoutError:
        print("timed out")

asyncio.run(main())
```

OUTPUT
Observed output:
timed out

14.3 Context variables

Context variables are designed for context-local state such as a request identifier. They propagate across asyncio task creation according to the current context. For this book's Python 3.13.5 baseline, a manually created thread should be treated as having its own context; copy and run the current context explicitly when inheritance is required. Python 3.14 adds a Thread(context=...) parameter and build-dependent default context inheritance, so later runtimes must be checked rather than assumed to behave exactly like 3.13. [PY-CONTEXT] [PY-THREAD-314]

Example 14.3 - Context variables propagate across asyncio tasks

PYTHON
```
import asyncio
from contextvars import ContextVar

request_id = ContextVar("request_id", default="missing")

async def worker():
    await asyncio.sleep(0)
    return request_id.get()

async def main():
    token = request_id.set("req-456")
    try:
        task = asyncio.create_task(worker())
        print(await task)
    finally:
        request_id.reset(token)

asyncio.run(main())
```

OUTPUT
Observed output:
req-456

Example 14.4 - Explicitly copy context into a new thread

PYTHON
```
from contextvars import ContextVar, copy_context
from threading import Thread

request_id = ContextVar("request_id", default="missing")

def worker():
    print(request_id.get())

token = request_id.set("req-thread")
try:
    ctx = copy_context()
    thread = Thread(target=ctx.run, args=(worker,))
    thread.start()
    thread.join()
finally:
    request_id.reset(token)
```

OUTPUT
Observed output:
req-thread

Every task needs an owner, a deadline or cancellation rule, and somewhere for failure to surface.

Otherwise background work becomes an incident that nobody observes.

Worked example: bound asynchronous concurrency

The semaphore makes the concurrency limit visible. Remove it only after measuring what the downstream system can accept.

```
PYTHON
import asyncio

async def fetch(item_id: int, limit: asyncio.Semaphore) -> str:
    async with limit:
        await asyncio.sleep(0.02)
        return f"item-{item_id}"

async def main() -> None:
    limit = asyncio.Semaphore(3)
    tasks = [fetch(i, limit) for i in range(6)]
    results = await asyncio.gather(*tasks)
    print(results)

asyncio.run(main())
```

Expected result: ['item-0', 'item-1', 'item-2', 'item-3', 'item-4', 'item-5']

Cancel the parent task and verify that cleanup still runs. Cancellation is part of the design, not an exotic error.

Trade-offs and failure modes

Async reliability depends on visible task lifetime, cancellation, and limits. A timeout does not prove the remote operation stopped; cancellation should normally clean up and propagate; blocking libraries still block the event loop; and ContextVar propagation into manually created threads must follow the runtime's documented behavior. [PY-CONTEXT] [PY-THREAD-314]

- Creating coroutine objects without awaiting or scheduling them.
- Calling blocking libraries inside the event-loop thread.
- Launching unbounded tasks without semaphore, queue, or worker limits.
- Swallowing CancelledError or leaving sibling-failure behavior undefined.

14.4 TaskGroup and structured lifetime

A TaskGroup owns the tasks created inside its block. Exiting waits for completion; when one task fails, sibling handling and grouped exceptions follow structured rules rather than leaving orphan tasks in the event loop. This makes task lifetime visible in the surrounding code.

Background work still requires an owner. Creating a task and dropping the reference does not define shutdown, error reporting, or retry behavior.

```
PYTHON
import asyncio

async def worker(name: str) -> str:
    await asyncio.sleep(0.01)
    return name

async def main() -> None:
    async with asyncio.TaskGroup() as group:
        tasks = [group.create_task(worker(name)) for name in ("A", "B")]
    print([task.result() for task in tasks])
```

```
asyncio.run(main())
```

14.5 Layered timeouts and backpressure

A connect timeout, read timeout, connection-pool timeout, and overall workflow deadline control different waits. One generous timeout around the whole coroutine can leave individual resource pools exhausted. Configure limits at the layer that owns each wait.

Concurrency also needs a bound. A semaphore or queue prevents one incoming batch from creating unbounded tasks and overwhelming the downstream system.

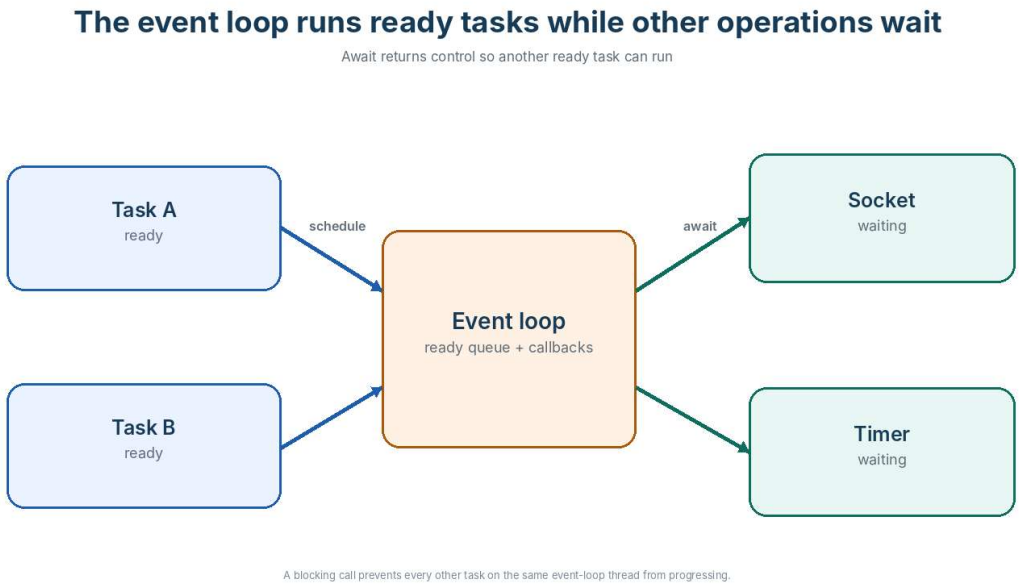

Figure 14.1 - The event loop advances ready tasks while sockets and timers wait.

Build: own task lifetime and cancellation

Own every task inside a clear lifetime. Add a timeout and one failure, then explain what happens to sibling tasks.

57. Write a sequential async function that awaits six simulated I/O operations one after another and measure it.
58. Schedule the operations concurrently and enforce a maximum of three in flight with a semaphore.
59. Add an overall timeout and verify that cleanup runs when the deadline expires.
60. Attach a ContextVar correlation ID and observe it in child tasks and in work delegated with asyncio.to_thread.
61. Create one failing task and define the intended behavior for sibling tasks using TaskGroup or gather.

Async evidence

- Concurrency bound
- Timeout behavior
- Cancellation cleanup
- Context propagation result
- Sibling-failure policy

Practice

62. Add a third coroutine to Example 14.1 and compare sequential and concurrent elapsed time.
63. Modify Example 14.2 to perform cleanup in finally.
64. Run a blocking time.sleep inside an async function and observe its effect.

PART IV - PRODUCTION SYSTEMS

Controlled metaprogramming, API contracts, reproducible packages, and secure operations.

Chapter 15 - Descriptors, Metaclasses, and Plugins

OPTIONAL ADVANCED CHAPTER — SAFE TO SKIP ON A FIRST READING
Continue directly to Chapter 16 if your goal is application development. Return here when descriptors, class-creation hooks, or a real plugin boundary appear in code you need to understand or extend.

Understand Python's advanced hooks, then prefer simpler mechanisms when they suffice.

Learning objectives

- explain the descriptor protocol
- recognize valid metaclass use cases
- avoid unsafe dynamic execution
- design a small plugin boundary

Frameworks sometimes need reusable hooks around attribute access, class creation, or plugin discovery. These mechanisms are worth learning because Python uses them internally - not because application code should reach for them first.

Properties, hooks, and class creation

Descriptors and metaclasses are easier to understand after seeing the ordinary mechanisms they generalize. Dunder methods connect user-defined objects to Python operations. A property controls one attribute. A class decorator transforms a class after creation. __init_subclass__ runs when a subclass is defined.

```
class Request:
    def __init__(self, request_id, quantity):
        self.request_id = request_id
        self.quantity = quantity

    def __repr__(self):
        return f"Request({self.request_id!r}, {self.quantity!r})"

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        value = int(value)
        if value <= 0:
            raise ValueError("quantity must be positive")
        self._quantity = value

print(Request("REQ-1", 2))
```

Use the simplest hook that solves the repeated problem. One validated field usually needs a property. Many identical fields across many classes may justify a descriptor. A family-wide class creation rule may justify __init_subclass__ or, rarely, a metaclass.

15.1 Descriptors

A descriptor is an object that participates in attribute access through methods such as __get__, __set__, and __delete__. Functions, properties, class methods, and many framework fields rely on descriptors. A custom descriptor is appropriate when the same attribute behavior must be reused across many classes.

Example 15.1 - A descriptor controls attribute access

PYTHON

```
class PositiveNumber:
    def __set_name__(self, owner, name):
        self.private_name = f"_{name}"

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return getattr(instance, self.private_name)

    def __set__(self, instance, value):
        if value <= 0:
            raise ValueError("value must be positive")
        setattr(instance, self.private_name, value)

class Product:
    price = PositiveNumber()

    def __init__(self, price):
        self.price = price

product = Product(99)
print(product.price)
```

OUTPUT
Observed output:

```
99
```

15.2 Metaclasses

A metaclass creates classes. The default metaclass is type. Custom metaclasses can validate class definitions, register subclasses, or alter namespaces at class creation time. They are rarely justified in application code because class decorators, __init_subclass__, and ordinary composition are easier to understand.

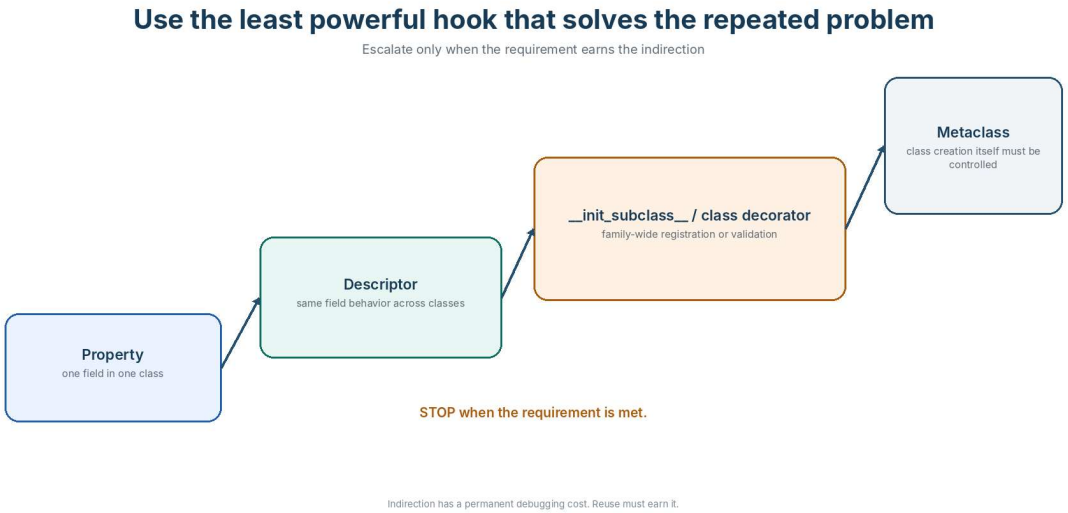

Figure 15.1 - Escalate from properties to metaclasses only when the requirement earns the indirection.

Choose the least powerful hook that solves the repeated problem. A normal registry, class decorator, or __init_subclass__ is easier to debug than a metaclass when either will do.

Example 15.2 - A minimal metaclass

```
PYTHON
class RequireVersion(type):
    def __new__(mcls, name, bases, namespace):
        if name != "Plugin" and "version" not in namespace:
            raise TypeError("plugins must define version")
        return super().__new__(mcls, name, bases, namespace)

class Plugin(metaclass=RequireVersion):
    pass

class ReportPlugin(Plugin):
    version = "1.0"

print(ReportPlugin.version)
```

OUTPUT
Observed output:
1.0

15.3 eval and exec are not sandboxes

eval executes an expression and exec executes statements. Passing restricted globals or removing __builtins__ does not create a reliable sandbox for hostile input. Object traversal and implementation details can expose dangerous capabilities. For structured data, use json. For Python literals, use ast.literal_eval. For a domain language, write or use a parser.

Example 15.3 - Safer literal parsing

```
PYTHON
from ast import literal_eval

value = literal_eval("{'retries': 3, 'enabled': True}")
print(value["retries"])
```

OUTPUT
Observed output:
3

15.4 Plugin boundaries

A plugin system needs discovery, a stable interface, lifecycle rules, error isolation, and version compatibility. Dynamic import is only one small part. A simple registry is often enough inside one application. Package entry points are appropriate when independently distributed packages must register plugins.

Example 15.4 - Simple plugin registration without exec

```
PYTHON
PLUGINS = {}

def register(name):
    def decorator(func):
        PLUGINS[name] = func
        return func
    return decorator

@register("uppercase")
def uppercase(value):
    return value.upper()

print(PLUGINS["uppercase"]("plugin"))
```

OUTPUT
Observed output:

PLUGIN

Controlled metaprogramming boundaries

Descriptors explain ordinary Python behavior before they justify custom framework machinery: functions, properties, class methods, and static methods all rely on descriptor behavior. Write a custom descriptor only when the same attribute rule is genuinely reused across classes.

Data descriptors implement __set__ or __delete__ and take precedence over instance state; non-data descriptors can be shadowed by an instance attribute. For one field in one class, a property is usually clearer.

Metaclasses control class creation, but __init_subclass__ and class decorators solve many registration and validation needs with less machinery. Use a custom metaclass only when the rule genuinely belongs to creation of an entire class family.

- Prefer property, class decorators, and __init_subclass__ before custom metaclasses.
- Keep plugin contracts narrow and versioned.
- Load only trusted code into the application process.
- Treat eval and exec as arbitrary code execution, not as a configurable calculator.

Worked example: reusable validation without hidden execution

The descriptor is justified here because the same rule is reusable. For one field in one class, a property would be clearer.

PYTHON

```
class PositiveInteger:
    def __set_name__(self, owner, name):
        self._name = "_" + name

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return getattr(instance, self._name)

    def __set__(self, instance, value):
        value = int(value)
        if value <= 0:
            raise ValueError("value must be positive")
        setattr(instance, self._name, value)

class RequestLine:
    quantity = PositiveInteger()

    def __init__(self, quantity):
        self.quantity = quantity

line = RequestLine(3)
print(line.quantity)
```

Expected result: 3

Replace the descriptor with a property and compare the amount of machinery. Reuse must earn the indirection.

Trade-offs and failure modes

Removing __builtins__ does not make eval or exec safe. For configuration, use a structured format and explicit schema; ast.literal_eval is limited to Python literal structures and is not a general expression sandbox.

Plugin systems expand both capability and attack surface. Entry points discovered through importlib.metadata provide a standard packaging-based registration mechanism, but importing a

plugin executes its module-level code. The host should define compatibility rules, isolate failures, validate plugin metadata, and decide whether plugins are trusted, signed, sandboxed in another process, or prohibited.

Metaprogramming earns its cost only when the abstraction is stable, documented, and observable. If readers cannot locate where behavior came from, debugging cost rises quickly.

- Using a metaclass where __init_subclass__ would be sufficient.
- Writing a descriptor without implementing instance is None handling.
- Hiding ordinary validation behind several levels of dynamic registration.
- Passing user input to eval or exec after superficial sanitization.
- Loading untrusted plugins into the main process.
- Failing the entire application because one optional plugin cannot load.

Build: choose the least powerful metaprogramming hook

Solve the same validation problem with a property, descriptor, __init_subclass__, and metaclass. Keep only the simplest version that meets the requirement.

65. Implement a reusable descriptor that validates positive quantities and records its field name through __set_name__.
66. Replace the descriptor with a property in a one-field class and compare readability and reuse.
67. Use __init_subclass__ to register subclasses by a declared plugin_name without introducing a metaclass.
68. Load one trusted plugin by module path and handle import or contract failure explicitly.
69. Demonstrate ast.literal_eval with safe literal input and reject an expression that attempts a function call.

Complexity justification

- Four implementations
- Readability comparison
- Failure isolation test
- Plugin trust decision
- The mechanism you kept and why

Practice

70. Extend PositiveNumber so it accepts zero when configured.
71. Replace the metaclass example with __init_subclass__.
72. Add version metadata and error handling to the plugin registry.

Chapter 16 - Building APIs with FastAPI and Flask

Design stable HTTP contracts and keep framework code at the boundary.

Learning objectives

- define an HTTP API contract
- validate request bodies
- distinguish sync and async routes
- compare FastAPI and Flask without hype

The request workflow becomes useful to other programs when it has a stable HTTP contract. Methods, schemas, status codes, authentication, idempotency, and errors matter more to clients than the framework behind the route.

16.1 HTTP contracts

A useful API defines resources, methods, status codes, request and response schemas, authentication, error formats, and idempotency expectations. A framework can generate documentation, but it cannot decide the business contract for you.

> HTTP note: 400 and 422 are not interchangeable labels with one universal framework rule. RFC 9110 defines 400 for a request the server will not process because of a perceived client error, while FastAPI's default request-validation handling commonly uses 422 and can be overridden. Choose the mapping as part of the public API contract and test it. A 401 response means the request lacks valid authentication credentials for the target resource; 403 is the normal choice when the caller is authenticated but not permitted. [HTTP-9110] [FASTAPI-ERRORS]

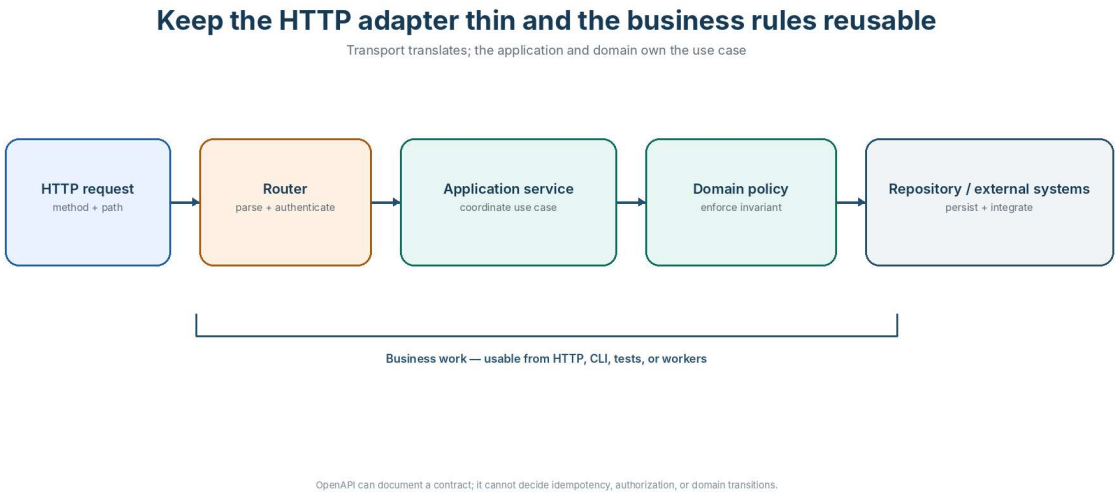

Figure 16.1 - A thin API adapter validates transport data and delegates one use case.

OpenAPI can document a broken contract perfectly. Decide duplicate-request behavior, authorization, conflicts, and error semantics before polishing generated documentation.

16.2 FastAPI

FastAPI uses Python type hints, Pydantic models, Starlette, and OpenAPI generation. Async routes are useful when the implementation awaits async dependencies. Declaring a route async does not make blocking database or network code non-blocking. [FASTAPI-ASYNC]

Example 16.1 - FastAPI endpoint with validation

PYTHON

```
from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field

app = FastAPI()

class EquipmentRequest(BaseModel):
    item: str = Field(min_length=2, max_length=80)
    quantity: int = Field(ge=1, le=20)

@app.post("/requests", status_code=201)
async def create_request(payload: EquipmentRequest):
    if payload.item.lower() == "forbidden":
        raise HTTPException(status_code=400, detail="item is not allowed")
    return {"status": "submitted", **payload.model_dump()}

client = TestClient(app)
response = client.post("/requests", json={"item": "Monitor", "quantity": 2})
print(response.status_code)
print(response.json()["status"])
```

OUTPUT

```
Observed output:
201
submitted
```

16.3 Flask

Flask is a minimal WSGI framework with a mature extension ecosystem. It offers more architectural freedom and therefore requires more explicit decisions around validation, schemas, and project structure. Minimal does not mean unsuitable for production; it means the application owns more choices. Flask supports async views, but under its normal WSGI model each request still occupies one worker for its request/response cycle. [FLASK-ASYNC]

Example 16.2 - Minimal Flask application

PYTHON

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.get("/health")
def health():
    return jsonify(status="ok")

if __name__ == "__main__":
    app.run(debug=True)
```

Concern	FastAPI	Flask
Data validation	Integrated through Pydantic	Choose an extension or validation library
API documentation	Automatic OpenAPI by default	Requires an extension or manual setup
Async model	ASGI-native	Traditionally WSGI; async support has constraints
Best fit	Typed APIs and async-capable	Flexible web applications and teams

	services	preferring explicit assembly

API contracts before framework code

An API contract includes method, path, request schema, response schema, status codes, authentication, idempotency, and error semantics. The framework routes a request, but the contract determines whether independent clients can rely on the service. A POST that sometimes creates a duplicate record after a timeout is not fixed by better documentation; it needs an idempotency design and a stable database constraint.

HTTP methods carry expectations. GET should be safe and should not change business state. PUT normally replaces or upserts a resource at a known identifier and is expected to be idempotent. PATCH applies a partial change whose semantics must be defined. POST commonly creates a subordinate resource or triggers a command. Status codes should distinguish invalid input, missing resources, conflicts, authentication failures, authorization failures, and server errors.

FastAPI combines ASGI routing with Pydantic validation and OpenAPI generation. Type hints improve the contract but do not automatically create good domain boundaries. Route functions should translate HTTP input into application commands, call a service, and translate domain outcomes into HTTP responses. Database queries and approval rules do not belong directly inside every endpoint.

- Define a stable error envelope with machine-readable code, message, and correlation ID.
- Keep authentication separate from authorization and enforce both at the correct boundary.
- Use dependency injection for resources and policies, not as a container for hidden global state.
- Test the public contract, including failures and generated schemas.

Worked example: an API boundary that stays thin

The route validates transport data and delegates. It does not become the domain model, repository, and notification system at once.

PYTHON

```
from fastapi import APIRouter, HTTPException, status
from pydantic import BaseModel, Field

router = APIRouter(prefix="/requests", tags=["equipment requests"])

class RequestCreate(BaseModel):
    employee_id: str = Field(min_length=1, max_length=40)
    item_type: str = Field(min_length=1, max_length=40)
    quantity: int = Field(ge=1, le=20)

@router.post("", status_code=status.HTTP_201_CREATED)
async def create_request(payload: RequestCreate) -> dict:
    if payload.item_type not in {"laptop", "monitor", "keyboard"}:
        raise HTTPException(status_code=422, detail="unsupported item type")
    return {"request_id": "REQ-1001", "status": "submitted"}
```

Expected result: Syntax-checked FastAPI route pattern; runtime behavior requires FastAPI and an ASGI test client.

Change the domain rule without touching the route. That is the boundary this example is trying to protect.

Trade-offs and failure modes

Flask provides fewer built-in opinions and remains effective when an application deliberately selects its validation, serialization, and documentation tools. An application factory avoids import-time global setup and makes configuration and testing easier. Blueprints organize routes, but they do not

create domain separation by themselves. The same service-layer boundary applies in both Flask and FastAPI.

Validation has layers. Schema validation checks types, ranges, and shape. Domain validation checks business rules such as allowed equipment, employee eligibility, and state transitions. Persistence validation enforces uniqueness and referential integrity. Authorization decides whether the authenticated actor may perform the operation. Mixing these layers into one enormous schema validator makes error handling and reuse difficult.

API observability should include request duration, status, route template, correlation ID, and stable business identifiers where safe. Never record passwords, access tokens, or complete sensitive payloads. Metrics need bounded labels; using raw request IDs as metric labels can create unbounded cardinality and operational failure.

- Returning HTTP 200 for every outcome, including errors.
- Putting database sessions and business rules directly into route functions.
- Trusting client-provided employee or role claims without authenticated identity mapping.
- Retrying create requests without an idempotency key or uniqueness constraint.
- Exposing internal exception messages and stack traces to clients.
- Logging full request payloads that contain sensitive information.

16.4 HTTP methods and status semantics

Status codes are part of the API contract. Returning 200 for every outcome forces clients to parse private message text.

Operation	Method	Typical success	Important property
Create request	POST	201 Created	May need idempotency protection
Read request	GET	200 OK	Safe and normally idempotent
Replace resource	PUT	200/204	Intended to be idempotent
Partial update	PATCH	200/204	Semantics must be defined
Delete request	DELETE	204	Authorization and state rules matter

16.5 Request and response models

Separate create, update, and response models. A create payload should not let the client assign server-owned fields such as approval status, audit timestamps, or approver identity. A response model can expose computed or read-only values.

Pydantic validates the declared shape at the HTTP boundary. Domain constructors should still protect invariants when objects can be created from tests, workers, or other adapters.

PYTHON
```
from pydantic import BaseModel, ConfigDict, Field

class RequestCreate(BaseModel):
    model_config = ConfigDict(extra="forbid")
    employee_id: str = Field(min_length=1, max_length=40)
    item_type: str = Field(min_length=1, max_length=80)
    quantity: int = Field(ge=1, le=20)

class RequestView(RequestCreate):
```

```
    request_id: str
    status: str
```

16.6 Routers and dependencies

Routers group endpoints by resource or capability. Dependencies provide request-scoped collaborators such as authenticated identity, database connections, or application services. Keep endpoint functions thin: parse the HTTP contract, call the service, and translate the result.

A dependency is not a reason to hide every object behind framework injection. Domain policy and pure helpers should remain ordinary Python.

```
PYTHON
from fastapi import APIRouter, Depends

router = APIRouter(prefix="/requests", tags=["requests"])

def get_service():
    ...

@router.get("/{request_id}")
async def read_request(request_id: str, service=Depends(get_service)):
    return await service.get(request_id)
```

16.7 Async database boundaries with asyncpg

An async endpoint benefits only when the libraries it awaits are non-blocking. Use a connection pool, acquire a connection for the shortest useful scope, parameterize SQL, and keep transactions explicit. Do not create a new database connection for every query. [ASYNCPG]

Repositories translate database rows and errors into application-level results. They should not decide HTTP status codes.

```
PYTHON
import asyncpg

async def fetch_request(pool: asyncpg.Pool, request_id: str):
    async with pool.acquire() as connection:
        row = await connection.fetchrow(
            "SELECT request_id, employee_id, status FROM equipment_request WHERE request_id = $1",
            request_id,
        )
    return None if row is None else dict(row)
```

16.8 Consistent API errors

Define one error shape containing a stable code, human-readable message, correlation ID, and optional field details. Do not expose raw tracebacks, SQL, credentials, or internal class names to the client.

Exception handlers should translate expected domain failures. Unexpected exceptions should be logged with the traceback at the owning boundary and returned as a generic server error.

```
PYTHON
from pydantic import BaseModel

class ApiError(BaseModel):
    code: str
    message: str
    correlation_id: str
    details: dict[str, str] | None = None
```

16.9 Testing the HTTP contract

API tests should assert status, response schema, headers, and side effects. Override dependencies or inject fakes so unit-level endpoint tests do not require a live database. Keep separate integration tests for the real SQL boundary.

Test failure contracts as carefully as success: malformed payload, missing resource, unauthorized actor, state conflict, and temporary dependency failure.

- Assert stable error codes, not only prose.
- Verify server-owned fields are rejected rather than accepted or silently ignored.
- Check idempotency behavior for repeated create commands.
- Exercise validation limits and unexpected extra fields.

16.10 Authentication, authorization, CORS, and rate limits

Authentication establishes identity. Authorization decides whether that identity may perform an action on a resource. Keep authorization close to the use case so it cannot be bypassed by another adapter.

CORS is a browser policy, not an authentication system. Rate limits reduce abuse and accidental overload but require a clear key, window, distributed-state strategy, and response contract.

> **Security controls work only when the boundary is enforced. A decorator is useful, but it does not replace object-level authorization, safe logging, input limits, or audit evidence.**

Build: protect an HTTP contract

Test the API as a client would see it: request schema, response schema, status, error code, and correlation ID.

73. Write an API contract for creating, reading, approving, and rejecting an equipment request before writing route code.
74. Define request and response schemas plus one consistent error envelope.
75. Implement one FastAPI router or Flask blueprint that delegates to an injected service.
76. Add tests for success, validation failure, missing resource, conflict, and forbidden approval.
77. Add an idempotency key or client request ID and define how duplicate submissions are handled transactionally.

What the API exercise should prove

- OpenAPI or route contract
- Success and failure tests
- Stable error envelope
- Authorization decision
- Blocking-I/O audit

Key takeaways

- Contract first: Define resources, schemas, status codes, and errors.
- FastAPI: Strong typed API ergonomics and OpenAPI generation.
- Flask: Minimal core and explicit architectural choices.
- Async caution: Blocking dependencies still block.

Practice

78. Add a GET route that retrieves a request by identifier.
79. Return a consistent error body for a missing request.
80. Write an API test for validation failure when quantity is zero.

Chapter 17 - Packaging and Dependency Management

Build installable artifacts that behave the same outside the source checkout.

Learning objectives

- structure an installable project
- use pyproject.toml
- isolate dependencies
- build and verify distributions

The code is ready for another developer only when it can be built and installed predictably outside the author's checkout. Packaging turns source files into an installable artifact with explicit metadata and dependencies; reproducible builds additionally require controlled inputs, dependency resolution, and build environments.

17.1 Use a src layout

A src layout keeps importable code under src/package_name and tests outside it. This reduces accidental imports from the working directory and makes tests exercise the installed package more honestly.

Path	Purpose
pyproject.toml	Build metadata, dependencies, and tool configuration
src/equipment_service/	Importable application package
tests/	Unit and integration tests
README.md	User-facing project instructions
.gitignore	Generated files and local secrets that must not be committed

17.2 pyproject.toml

pyproject.toml is the standard entry point for modern Python build configuration. The build-system table selects a backend. The project table contains package metadata and dependencies. Tool-specific tables configure formatters, type checkers, and test runners.

Example 17.1 - Modern pyproject.toml

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "equipment-service"
version = "0.1.0"
description = "A small equipment request service"
requires-python = ">=3.11"
dependencies = ["fastapi>=0.115", "uvicorn>=0.30"]

[project.optional-dependencies]
test = ["pytest>=8", "httpx>=0.27"]
```

17.3 Virtual environments and lock files

Use one isolated environment per project. pip, uv, Poetry, and other tools differ in workflow, but the requirement is the same: resolve dependencies deterministically, review updates, and reproduce the environment in CI. pip freeze is a snapshot, not a complete dependency-management strategy.

A lock file can reproduce a vulnerable dependency exactly. Reproducibility tells you what you installed; security review tells you whether you should keep it.

Example 17.2 - Create and activate a virtual environment

```
SHELL
python -m venv .venv
# Windows PowerShell
.venv\Scripts\Activate.ps1
# Linux or macOS
source .venv/bin/activate
```

17.4 Build before publishing

Build a source distribution and wheel, inspect their contents, install the wheel into a clean environment, run tests, and validate metadata with twine check. Do not publish by running setup.py directly; use python -m build with a modern backend.

Example 17.3 - Build and verify a package

```
SHELL
python -m pip install --upgrade build twine
python -m build
python -m twine check dist/*
```

Reproducible builds and releases

Reproducibility comes from one authoritative build configuration, an isolated dependency workflow, and testing the artifact that will actually be released. The src layout from Section 17.1 helps expose accidental repository-root imports, but the decisive check is a clean installation.

Keep build metadata and dependency declarations in one authoritative workflow so developers and CI resolve the same intended environment.

A wheel is a built distribution; a source distribution carries source and build metadata. Libraries commonly publish both, while internal applications may promote a wheel or container artifact. Build the release artifact once, verify it in a clean environment, and promote that same artifact through later environments rather than rebuilding it independently.

- Install and test the built artifact, not only the source checkout.
- Keep package metadata, version, and dependency declarations in one authoritative workflow.
- Use environment variables or secret stores for runtime secrets; never place real credentials in package files.
- Treat release notes, compatibility, and deprecation as part of the package contract.

Worked example: package metadata that can be built

The metadata is short, but every field has release consequences. Build the package before assuming the configuration is correct.

```
PYTHON
from importlib.metadata import PackageNotFoundError, version

def installed_version(distribution: str) -> str:
```

```
    try:
        return version(distribution)
    except PackageNotFoundError:
        return "not-installed"

print(installed_version("pip"))
```

Expected result: The installed pip distribution version is printed, or not-installed in an unusual environment.

Build a wheel and inspect it. Packaging configuration is not proven by looking plausible.

Trade-offs and failure modes

An __init__.py file marks a regular package and can expose a deliberate public API. Namespace packages can span multiple directories without __init__.py, but they solve a specific distribution problem and should not be introduced accidentally. Keep package initialization lightweight; importing a package should not connect to databases, start threads, or read environment-specific secrets.

Dependency ranges balance compatibility and reproducibility. Libraries usually avoid pinning every transitive dependency because doing so can conflict with applications. Applications commonly use lock files or constraints to reproduce a tested environment. Security updates complicate permanent pinning, so dependency automation and regular rebuilds are necessary. Reproducible does not mean frozen forever.

Versioning communicates compatibility only when the project defines what its public API includes. Semantic versioning can guide expectations, but schema changes, CLI changes, configuration changes, and behavior changes may all affect consumers. A release checklist should include tests, type checks, build, artifact inspection, vulnerability review, changelog, and rollback information.

- Testing only from the repository root and never installing the built wheel.
- Executing infrastructure setup from package import side effects.
- Publishing credentials, tokens, or internal URLs in configuration examples.
- Maintaining dependencies in several conflicting files without an authoritative source.
- Using an obsolete setup.py upload workflow instead of a modern build and twine process.
- Rebuilding different artifacts for each deployment environment.

17.5 Verify the built artifact

Install the wheel into a clean environment and run public imports, entry points, and tests against that installation. This exposes missing package data, bad metadata, and imports that worked only from the repository root.

Treat the installed artifact, not the source checkout, as the release candidate you are proving.

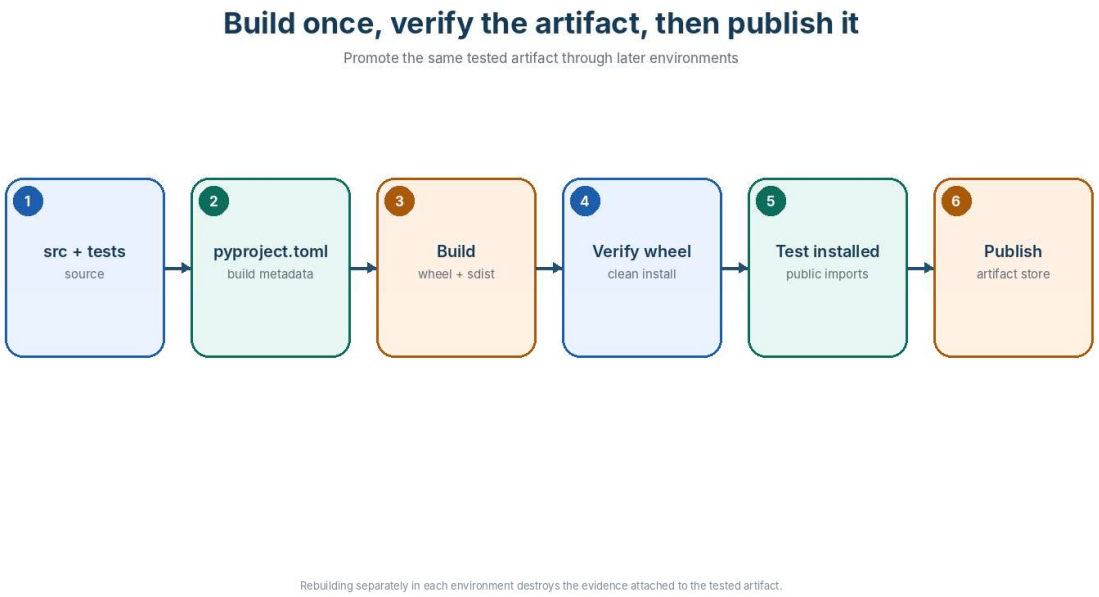

Figure 17.1 - Build once, verify the artifact in a clean environment, and then publish it.

17.6 Versioning and release evidence

A release should identify source commit, package version, dependency resolution, build environment, tests, and generated artifacts. Build once and promote the same artifact rather than rebuilding independently in every environment.

Semantic versioning can communicate intent, but compatibility still depends on the public contract. Deprecate before removal, document migration, and test supported versions.

- Tag immutable source.
- Generate wheel and source distribution in CI.
- Inspect package contents.
- Install the wheel into a clean environment.
- Publish checksums and provenance where required.

Build: produce an installable artifact

Create the package from a clean environment, install the built wheel elsewhere, and run a public entry point.

81. Create a small src-layout project with one package, one public function, tests, and pyproject.toml.
82. Build both wheel and source distribution with python -m build.
83. Create a fresh virtual environment, install the wheel, and run tests against the installed artifact.
84. Inspect the wheel contents and verify that tests, secrets, local data, and temporary files are excluded.
85. Draft a release checklist including version, changelog, build verification, signing or checksum, and rollback.

What the package exercise should prove

- Built wheel
- Clean-environment install
- Version metadata
- Dependency record
- Public import or entry-point test

Practice

86. Create the shown project structure.
87. Add a console-script entry point to pyproject.toml.
88. Build a wheel and inspect it with unzip -l or an archive viewer.

Chapter 18 - Deployment, Operations, and Security

Scope: this chapter explains production controls; the examples are not a certification of production readiness. Load, security, migration, backup, and incident procedures still require target-environment testing.

Treat deployment as an operating system around the application, not a packaging step.

Learning objectives

- externalize configuration
- protect secrets
- design a container responsibly
- select a deployment strategy
- apply secure input handling

Deployment adds concerns that do not appear when one developer runs one process locally: configuration ownership, secrets, health, capacity, migrations, rollback, and incident response.

18.1 Configuration and secrets

Configuration changes between environments; code should not. Read configuration from environment variables or an approved secret manager. Validate required settings at startup. Never commit real secrets to source control or place them in example files that developers may copy unchanged.

Example 18.1 - Read secrets from environment variables

PYTHON

```
import os

database_url = os.getenv("DATABASE_URL")
if not database_url:
    print("DATABASE_URL is not configured")
```

OUTPUT

```
Observed output:
DATABASE_URL is not configured
```

18.2 Input and dynamic execution

Treat all external input as untrusted. Validate shape, size, type, authorization, and business rules. Use parameterized database queries. Do not pass user input to eval, exec, shell=True, or dynamically constructed import paths. Removing built-ins is not a substitute for isolation. [OWASP-INPUT]

Example 18.2 - Do not build a sandbox with eval

PYTHON

```
import json

raw = '{"quantity": 2, "approved": false}'
payload = json.loads(raw)
print(payload["quantity"])
```

OUTPUT

```
Observed output:
2
```

18.3 Containers

A container should use a small maintained base image, install only required dependencies, avoid running as root when the service does not require those privileges, expose configuration through the environment, provide a health endpoint, and write logs to standard output. Pinning every dependency forever is not security; controlled upgrades and vulnerability review are. [DOCKER-BEST]

> **A container provides a packaging boundary and OS-level process-isolation mechanisms. It does not decide migrations, rollback, health semantics, secrets, capacity limits, or who responds when a release fails.**

Example 18.3 - A production-oriented Dockerfile

```
DOCKERFILE
FROM python:3.13-slim
WORKDIR /app
COPY pyproject.toml .
COPY src ./src
RUN pip install --no-cache-dir .
USER 10001
CMD ["uvicorn", "equipment_service.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

18.4 Release strategies

Strategy	How it works	Trade-off
Rolling	Replace instances gradually	Simple, but old and new versions coexist
Blue/green	Switch traffic between two complete environments	Fast rollback, higher infrastructure cost
Canary	Expose a small percentage of traffic first	Limits blast radius, requires strong observability

18.5 Production checklist

- Availability: health checks, timeouts, retries with limits, graceful shutdown, and capacity monitoring.
- Security: least privilege, secret rotation, dependency scanning, authorization checks, and audit logs.
- Observability: structured logs, metrics, traces, correlation identifiers, and actionable alerts.
- Recovery: tested backups, rollback procedures, incident ownership, and documented failure modes.

Worked example: configuration without embedded secrets

The example reads configuration from the environment. A real deployment must also define ownership, rotation, and failure behavior.

```
PYTHON
from dataclasses import dataclass
import os

@dataclass(frozen=True)
class Settings:
    environment: str
    log_level: str

    @classmethod
    def from_environment(cls) -> "Settings":
        environment = os.getenv("APP_ENV", "development")
        log_level = os.getenv("LOG_LEVEL", "INFO").upper()
        if log_level not in {"DEBUG", "INFO", "WARNING", "ERROR"}:
```

```
            raise ValueError("invalid LOG_LEVEL")
        return cls(environment=environment, log_level=log_level)

print(Settings.from_environment())
```

Expected result: Settings(environment='development', log_level='INFO') when no variables are set.

Set LOG_LEVEL to an unsupported value and confirm that startup fails clearly rather than continuing with an invalid configuration.

Trade-offs and failure modes

Production controls work as a system: release strategy must fit state compatibility, security requires layered controls, and capacity must be measured against real dependencies and limits. The recurring failures are uncontrolled secrets or privilege, health checks with the wrong semantics, schema changes that block rollback, unbounded observability labels, and scaling that simply moves the bottleneck into the database or memory limit.

- Embedding secrets in source, images, example configuration, or logs.
- Using liveness checks that fail because a temporary dependency is unavailable.
- Deploying schema changes that make the previous application version unable to run.
- Using unbounded metric labels or scaling workers beyond downstream capacity.

18.6 A hardened container baseline

A production image should be small enough to audit, run as a non-root user, contain only runtime dependencies, and avoid copying credentials or development caches. Pin the base image deliberately and rebuild when security updates require it.

Use a multi-stage build when compilers and build tools are not needed at runtime. A container is a packaging boundary, not a complete security boundary.

DOCKERFILE

```
FROM python:3.13-slim AS runtime
WORKDIR /app
RUN useradd --create-home appuser
COPY dist/equipment_service-*.whl /tmp/app.whl
RUN python -m pip install --no-cache-dir /tmp/app.whl && rm /tmp/app.whl
USER appuser
EXPOSE 8000
CMD ["uvicorn", "equipment.api:app", "--host", "0.0.0.0", "--port", "8000"]
```

18.7 CI/CD as a controlled promotion flow

A delivery pipeline should compile or build, run tests, scan dependencies and artifacts, publish an immutable artifact, deploy to a controlled environment, verify health, and record promotion. Manual approvals may be appropriate for regulated or high-risk transitions, but manual repetition is not a control.

Separate deployment from release when feature flags or traffic controls allow it. Rollback must be rehearsed and compatible with database changes.

Stage	Evidence
Build	Source revision, dependency resolution, artifact digest
Verify	Tests, lint, type checks, security scans
Deploy	Environment, configuration version, migration result
Observe	Health, error rate, latency, saturation

Stage	Evidence
Promote or rollback	Decision, actor, timestamp, resulting version

18.8 Observability and service objectives

Logs describe discrete events, metrics summarize behavior over time, and traces connect work across components. Health endpoints should distinguish process liveness from readiness to serve traffic. A service can be alive while unable to reach its required database.

Define a small set of service objectives: availability, latency, error rate, queue delay, or completion time. Alerts should indicate actionable risk to those objectives rather than every unusual log line.

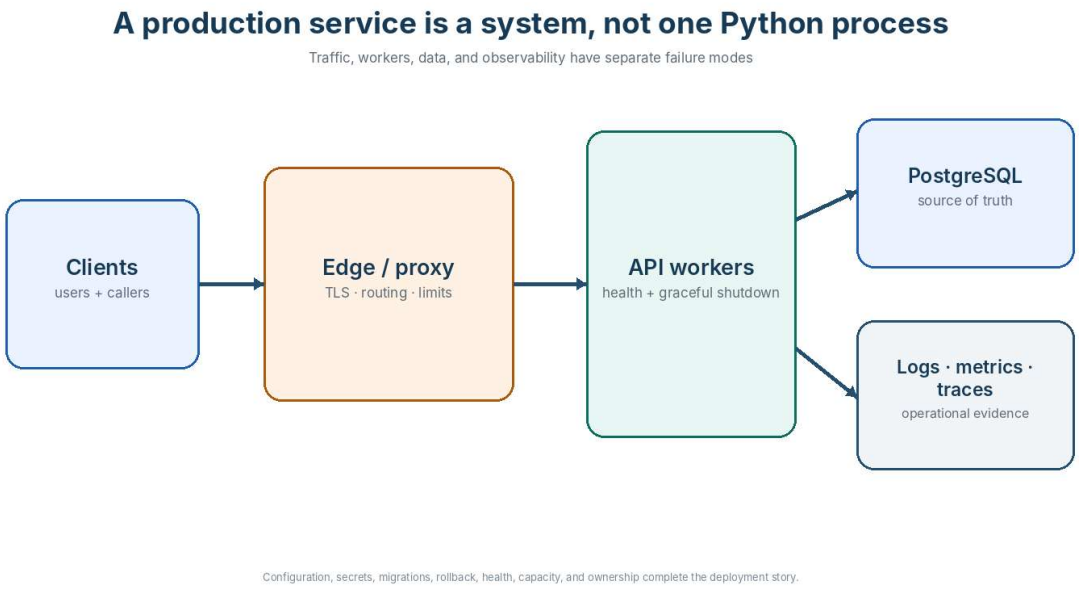

Figure 18.1 - A production service includes traffic control, workers, data, and observability.

18.9 Threat modeling the service

List assets, actors, trust boundaries, entry points, and likely abuse. For an internal equipment system, assets include employee data, approval authority, inventory, audit evidence, and credentials. Threats include privilege escalation, forged approvals, injection, data exposure, denial of service, and tampered dependencies.

Map each important threat to preventive, detective, and recovery controls. A threat model is useful only when it changes design, tests, monitoring, or operating procedures.

- Authenticate every caller.
- Authorize the action and resource.
- Parameterize database access.
- Validate and bound input.
- Protect secrets and logs.
- Record approval evidence.
- Test recovery and revocation.

18.10 Database migrations and rollback

Application rollback is unsafe when the deployed database schema cannot support the previous version. Use backward-compatible expand-and-contract migrations: add new structures, deploy code that can use both, migrate data, switch behavior, then remove old structures later.

Backups are not a rollback plan unless restore time and data-loss tolerance have been measured. Record migration versions and make failures visible before traffic reaches incompatible code.

> **Application code and schema evolve together. Design the migration and rollback path before deployment so the previous version can still run while the database is changing.**

Build: rehearse a controlled release

Promote one version through a local release rehearsal. Include configuration failure, health checks, logs, and rollback evidence.

89. Define one immutable application artifact and a separate environment configuration contract.
90. Write a Dockerfile plan that uses a non-root user, a small runtime image, and no embedded secrets.
91. Specify distinct liveness, readiness, and startup checks for the service.
92. Design a backward-compatible database migration that supports both old and new application versions during rollout.
93. Create a deployment runbook with prechecks, monitoring signals, rollback trigger, backup verification, and ownership.

What the release exercise should prove

- Release checklist
- Health/readiness evidence
- Redacted configuration
- Rollback rehearsal
- Threat-model note

Practice

94. Add a startup check for DATABASE_URL.
95. Write a threat model for the equipment request endpoint.
96. Explain what a health endpoint should and should not test.

PART V - CAPSTONE

Integrate the language, design, persistence, API, reliability, and operational controls into one service.

Chapter 19 - Capstone: Employee Equipment Request Service

> Scope: the capstone is an implementable reference architecture, not a claim of a fully deployed production system. Database, authentication, notification, deployment, load, and rollback behavior still require integration and operational tests.

Connect domain rules, persistence, HTTP, testing, and operations in one complete workflow.

Learning objectives

- model a workflow state
- protect state transitions
- separate domain and transport concerns
- define a phased implementation plan

The capstone assembles the decisions made throughout the book into one workflow. The goal is not a large architecture; it is a complete request path whose state, persistence, HTTP boundary, authorization, failure handling, and operations remain visible.

Beginner walkthrough: build one vertical slice first

The capstone becomes manageable when the first version is deliberately narrow. Begin with one equipment request, one valid transition, one in-memory repository, and one test. Add PostgreSQL, HTTP, authentication, notifications, and deployment only after the core behavior is visible and correct.

97. Create a Request data model with request_id, employee_id, item_type, quantity, and status.
98. Validate quantity and allowed item types when the object is created.
99. Implement approve() and reject() methods that protect valid state transitions.
100. Store requests in an in-memory dictionary behind a repository interface.
101. Write tests for create, approve, reject, repeated approval, and unknown request.
102. Expose the same service through a command-line script before adding a web framework.
103. Replace the in-memory repository with asyncpg while keeping domain tests unchanged.
104. Add FastAPI routes, request schemas, authentication, audit logging, and deployment controls.

```
from dataclasses import dataclass

@dataclass
class Request:
    request_id: str
    employee_id: str
    item_type: str
    quantity: int
    status: str = "submitted"

    def __post_init__(self):
        if self.quantity <= 0:
            raise ValueError("quantity must be positive")

    def approve(self):
```

```
        if self.status != "submitted":
            raise ValueError(f"cannot approve from {self.status}")
        self.status = "approved"

request = Request("REQ-1", "E-100", "laptop", 1)
request.approve()
print(request.status)
```

Layer	Beginner responsibility	Later production responsibility
Domain	State and transition rules.	Invariants, events, authorization inputs.
Service	Coordinate one use case.	Transactions, idempotency, audit coordination.
Repository	Save and retrieve requests.	asyncpg queries, locking, mapping, retries.
API	Convert HTTP input to service calls.	Authentication, schemas, errors, versioning.
Operations	Run the program.	Configuration, deployment, monitoring, recovery.

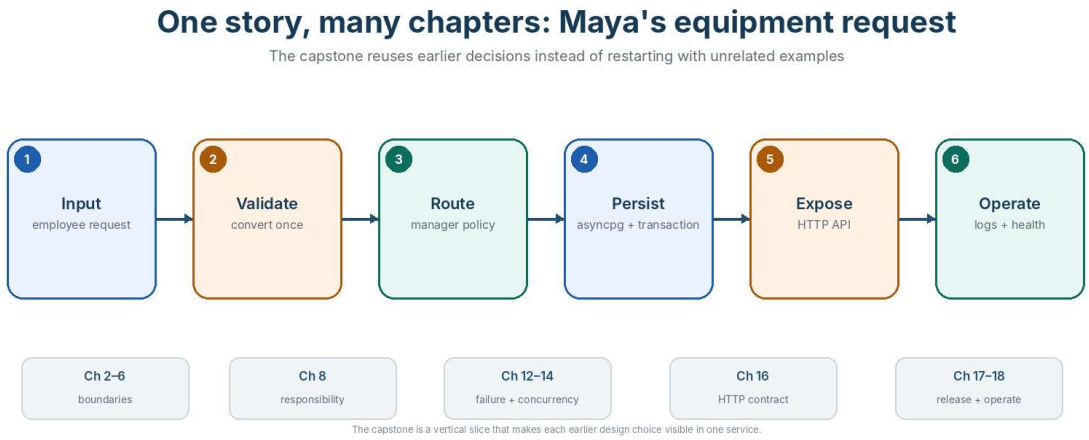

Figure 19.1 - The running story becomes the capstone architecture.

19.1 Scope

Employees submit requests for equipment such as laptops, monitors, keyboards, and software licenses. Managers or administrators approve or reject requests and record comments. PostgreSQL is the source of truth. The API exposes workflow operations; the React frontend presents role-appropriate views.

19.2 Domain model first

Teaching state machine. Begin with the deliberately small model used by the first vertical slice: SUBMITTED -> APPROVED or REJECTED. A decided request does not silently return to SUBMITTED. Example 19.1 intentionally uses this minimal model so transition invariants are clear before more workflow states are introduced.

This reuses Chapter 8's responsibility rule: the domain owns the invariant; transport and persistence layers must not bypass it.

Example 19.1 - Capstone domain model and workflow

PYTHON

```
from dataclasses import dataclass, field
from enum import StrEnum

class Status(StrEnum):
    SUBMITTED = "submitted"
    APPROVED = "approved"
    REJECTED = "rejected"

@dataclass
class EquipmentRequest:
    employee_id: str
    item: str
    quantity: int = 1
    status: Status = Status.SUBMITTED
    comments: list[str] = field(default_factory=list)

    def approve(self, comment=None):
        if self.status != Status.SUBMITTED:
            raise ValueError("only submitted requests can be approved")
        self.status = Status.APPROVED
        if comment:
            self.comments.append(comment)

request = EquipmentRequest("E-100", "Laptop")
request.approve("Budget confirmed")
print(request.status)
print(request.comments)
```

OUTPUT

```
Observed output:
approved
['Budget confirmed']
```

19.3 Suggested service boundaries

Layer	Responsibility	Example
Domain	State, rules, transitions	EquipmentRequest.approve()
Application service	Coordinates use cases and transactions	approve_request(request_id, actor)
Repository	Persists and loads aggregates	PostgreSQL request repository
API router	HTTP parsing, authentication, response mapping	POST /requests/{id}/approve
Frontend	Forms, status views, role-aware actions	Employee dashboard and manager queue

19.4 PostgreSQL schema outline

Table	Important columns	Purpose
users	id, employee_code, name, role	Employees, managers, administrators
equipment_requests	id, requester_id, item, quantity, status, created_at, updated_at	Workflow source of truth
request_comments	id, request_id, author_id, comment, created_at	Decision rationale and discussion
audit_events	id, request_id, actor_id, event_type, payload, created_at	Traceability and evidence

19.5 API outline

Method	Path	Purpose
POST	/requests	Submit a request
GET	/requests/me	List the current employee's requests
GET	/requests/pending	List requests awaiting decision
POST	/requests/{id}/approve	Approve a submitted request
POST	/requests/{id}/reject	Reject a submitted request
POST	/requests/{id}/comments	Add a comment

19.6 Phased implementation

105. Implement the domain model and transition tests.

Build one submit-to-approve path before filling every folder. A working slice exposes bad boundaries faster than a directory full of interfaces.

106. Create PostgreSQL tables and an async repository using asyncpg.
107. Add FastAPI routers, Pydantic schemas, authentication hooks, and consistent errors.
108. Build the React employee dashboard and manager approval queue.
109. Add structured logging, audit events, correlation identifiers, and integration tests.
110. Containerize, configure CI, deploy to a test environment, and exercise rollback.

Note: Design reality: The capstone is intentionally modest. Adding AI, notifications, asset inventory, procurement, or policy engines before the core workflow is correct would increase surface area without proving the foundation.

Turning the capstone into a complete service

Production extension. Once the teaching state machine is correct, a real deployment may extend the workflow with states such as MANAGER_REVIEW, FULFILLED, and CANCELLED. These states extend rather than contradict the earlier model. Every added transition should name the actor, validate the current state, record required reasoning, update atomically, and append an audit event. Direct status assignment from route code still bypasses the workflow contract.

The domain model should distinguish identity, value, and persistence. Request ID and employee ID identify entities. Item type, quantity, justification, and status describe state. Approval policy determines routing. A repository stores and retrieves requests without owning business decisions. An application service coordinates policy, repository, audit, and notification boundaries under one transaction strategy.

PostgreSQL remains the source of truth. Constraints should enforce non-null fields, valid quantities, unique identifiers, and foreign-key relationships. A version column or conditional update can prevent lost updates when two reviewers act concurrently. Audit events should be append-only from the application's perspective and should include request ID, event type, actor, timestamp, correlation ID, and safe metadata.

- Model commands and transitions explicitly instead of exposing unrestricted status updates.
- Use idempotency for create and approval commands that clients may retry.
- Keep notifications outside the core transaction through an outbox or reliable event process when delivery matters.
- Test both normal flow and conflicting, repeated, unauthorized, and partially failed operations.

Worked example: one vertical slice through the capstone

This vertical slice is intentionally narrow: one command enters, one rule runs, one record is stored, and one response leaves.

```
PYTHON
from dataclasses import dataclass
from enum import StrEnum

class Status(StrEnum):
    SUBMITTED = "submitted"
    APPROVED = "approved"
    REJECTED = "rejected"

@dataclass
class EquipmentRequest:
    request_id: str
    status: Status = Status.SUBMITTED

    def approve(self) -> None:
        if self.status is not Status.SUBMITTED:
            raise ValueError(f"cannot approve from {self.status}")
        self.status = Status.APPROVED

request = EquipmentRequest("REQ-1001")
request.approve()
print(request.status)
```

Expected result: approved

Trace the request ID through every layer. If it disappears, debugging the finished service will be expensive.

Trade-offs and failure modes

Transaction boundaries need deliberate design. Saving a request and sending an email in one function does not make the operations atomic because an SMTP server cannot participate in the PostgreSQL transaction. If the database commits and email fails, retrying the whole command may duplicate the state change. An outbox record written in the same database transaction can be delivered asynchronously with retries and deduplication.

Role-based behavior should not be implemented only by hiding buttons in the frontend. The backend must authenticate the actor and authorize the action against the resource. An employee may view their own requests, a manager may review requests for their reporting scope, and an administrator may manage catalog or fulfillment actions. Object-level authorization is essential because knowing a request ID must not grant access.

The capstone should remain buildable. Do not start with Kubernetes, event streaming, and dozens of microservices. Begin with a modular monolith: React frontend, FastAPI backend, PostgreSQL, explicit services and repositories, structured logs, and tests. Add background delivery, caching, or distributed components only when requirements and measured load justify them.

- Allowing arbitrary PATCH updates to status without transition validation.
- Sending notifications inside the database transaction and assuming atomic delivery.
- Relying on frontend role checks without backend authorization.
- Creating duplicate requests after client timeout and retry.
- Losing one manager decision when concurrent updates overwrite each other.
- Building distributed infrastructure before the workflow and data model are stable.

19.7 Component architecture

The UI calls a FastAPI adapter. The adapter validates transport data and invokes application services. Domain objects enforce state transitions. Repositories persist requests and audit events through asyncpg. A notification worker reacts to committed events rather than sending email inside an open database transaction.

The boundary is the combination of Chapters 8 and 16: composition keeps collaborators replaceable while the API remains a thin transport adapter.

Each boundary has one reason to change and one clear failure contract.

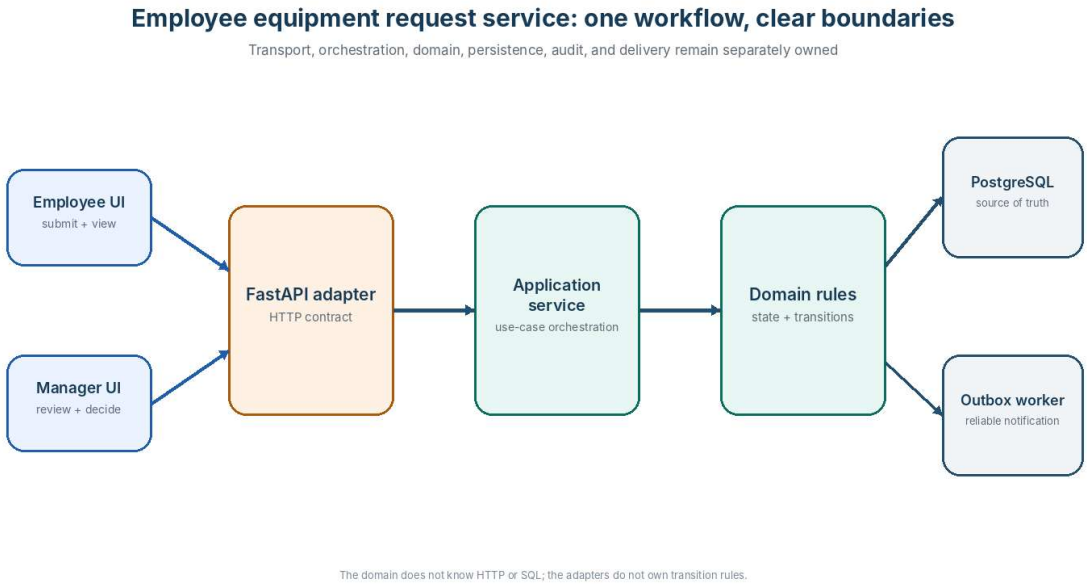

Figure 19.2 - The capstone keeps transport, orchestration, domain, persistence, audit, and notification responsibilities visible.

19.8 Request state machine

State transitions are methods or policies, not arbitrary string assignments. Every transition checks the current state, actor authority, required reason, and resulting audit event. Invalid transitions raise a domain conflict rather than silently overwriting history.

Chapter 12's failure-contract idea now becomes workflow behavior: invalid transitions are named conflicts rather than silent status overwrites.

Figure 19.3 illustrates one production extension of the teaching state machine: it adds manager review and fulfillment around the core approval/rejection decision. Other deployments may add cancellation, return, partial fulfillment, or escalation; each added state increases authorization, testing, migration, and reporting complexity.

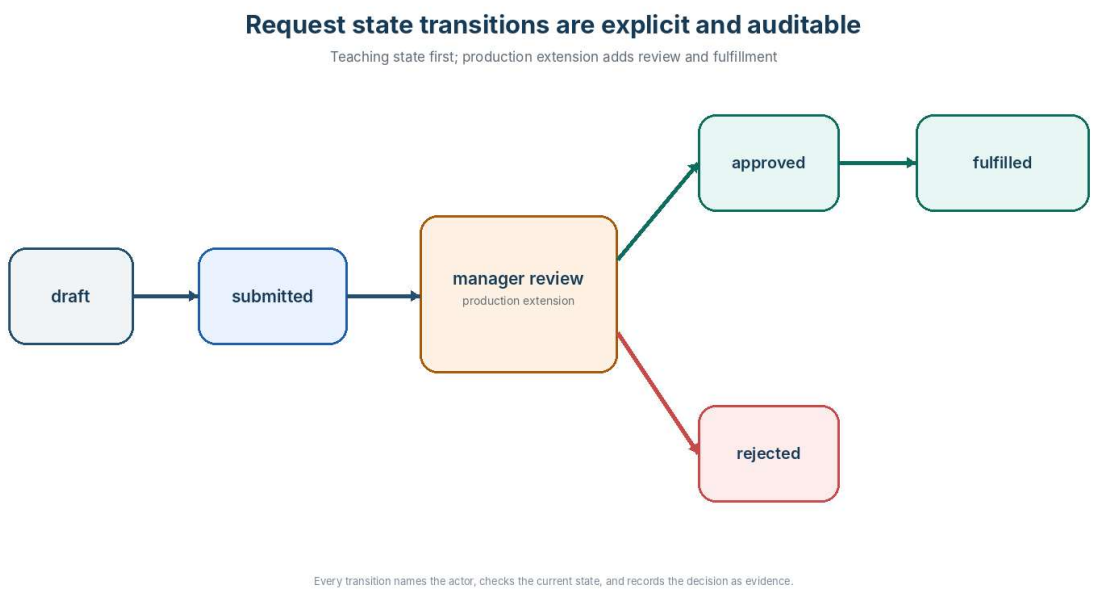

Figure 19.3 - The request state machine permits named transitions and rejects arbitrary status assignment.

19.9 asyncpg pool lifecycle

Create the pool during application startup and close it during shutdown. Configure minimum and maximum size from validated settings. The pool is shared infrastructure; individual requests acquire and release connections for short scopes.

This applies Chapter 14's explicit async resource lifetime and Chapter 16's asyncpg boundary: share the pool, but keep each acquired connection short-lived.

Do not print the database URL because it may contain credentials. Log a safe host or database identifier only when operationally required.

PYTHON
```
import asyncpg

async def create_pool(database_url: str) -> asyncpg.Pool:
    return await asyncpg.create_pool(
        dsn=database_url,
        min_size=1,
        max_size=10,
        command_timeout=10,
    )
```

19.10 Repository implementation

The repository owns SQL and row mapping. Parameter placeholders prevent injection and let the driver transmit values separately from SQL text. A transaction should include the state update and audit insert so the operational record and evidence cannot diverge.

The repository turns Chapters 12 and 16 into persistence rules: parameterize SQL, translate database outcomes at the boundary, and preserve concurrency evidence.

Use optimistic concurrency or a state predicate in the UPDATE statement to prevent two reviewers from approving the same submitted request concurrently.

SQL
```
UPDATE equipment_request
SET status = $2, reviewed_by = $3, reviewed_at = now(), version = version + 1
WHERE request_id = $1 AND status = 'submitted' AND version = $4
RETURNING request_id, status, version;
```

19.11 Application service orchestration

The service loads the request, authorizes the actor, asks the domain object to transition, persists the new state and audit event in one transaction, and publishes an outbox event for later notification. It does not know HTTP status codes or HTML templates.

This is Chapter 8's constructor-injection design in the capstone: required collaborators are visible, replaceable, and independently testable.

Required collaborators are constructor arguments, making the service easy to test with fakes.

PYTHON
```
class ReviewRequestService:
    def __init__(self, repository, authorizer, clock):
        self._repository = repository
        self._authorizer = authorizer
        self._clock = clock

    async def approve(self, command):
        request = await self._repository.get(command.request_id)
        self._authorizer.require_manager(command.actor, request)
        event = request.approve(command.actor.user_id, self._clock.now())
        await self._repository.save_with_event(request, event)
        return request
```

19.12 FastAPI router

The router receives transport values, resolves the authenticated actor and service, and maps the domain result to a response model. Domain conflicts and missing records are translated by central exception handlers, not repeated in every route.

This is the Chapter 16 contract-first rule applied directly: the route handles HTTP concerns and delegates the use case instead of becoming the business layer.

Use a command model for the approval reason or idempotency key when the contract requires it.

PYTHON
```
@router.post("/{request_id}/approve", response_model=RequestView)
async def approve_request(
    request_id: str,
    body: ReviewBody,
    actor: Actor = Depends(current_actor),
    service: ReviewRequestService = Depends(review_service),
):
    command = ApproveRequest(request_id=request_id, actor=actor, reason=body.reason)
    return await service.approve(command)
```

19.13 Transport schemas

Create and response schemas should differ. The server assigns request_id, status, timestamps, and review fields. Use field constraints for basic shape and domain constructors for business invariants. Reject unknown fields when accepting them would hide client mistakes.

Response serialization should not leak internal database columns or security-sensitive notes.

PYTHON
```
from pydantic import BaseModel, ConfigDict, Field

class RequestCreate(BaseModel):
    model_config = ConfigDict(extra="forbid")
    item_type: str = Field(min_length=1, max_length=80)
    quantity: int = Field(ge=1, le=20)
    justification: str = Field(min_length=5, max_length=1000)
```

19.14 Error and response mapping

The client receives stable public codes. Internal logs retain diagnostic evidence without exposing it over HTTP.

Domain condition	API response	Log behavior
Validation failure	422 with field details	Info; no traceback
Request missing	404	Info with request_id
Invalid transition	409	Warning if repeated or suspicious
Permission denied	403	Security event without sensitive payload
Database unavailable	503	Error with traceback and correlation_id
Unexpected defect	500 generic body	Error with traceback; alert by policy

19.15 Unit testing domain transitions

Domain tests should construct requests directly and assert transition result, state, and event. They do not need FastAPI, asyncpg, or mocks. Cover every valid edge and every rejected edge in the state machine.

Use a fixed clock and explicit actor IDs so output is deterministic.

- submitted -> approved by authorized manager
- submitted -> rejected with non-blank reason
- approved cannot be rejected
- rejected cannot be fulfilled
- employee cannot review own request unless policy explicitly allows it

19.16 Integration testing PostgreSQL

Integration tests should run migrations against an isolated database, insert controlled records, execute repository methods, and verify transaction behavior. Roll back per test or recreate the schema so tests do not depend on execution order.

Test optimistic concurrency by attempting two updates with the same version and asserting that only one succeeds.

> **A repository test that mocks asyncpg verifies your wrapper, not PostgreSQL behavior. Keep unit tests, but add integration tests for SQL, types, constraints, and transactions.**

19.17 Idempotency and duplicate commands

Clients retry when connections fail, and they may not know whether the server committed the first attempt. A create or notification operation can duplicate effects unless the command carries an idempotency key stored with the result.

The retry warning from Chapter 9 meets the API contract from Chapter 16 here: retries are safe only when duplicate effects are controlled transactionally.

Define the key scope, retention period, request fingerprint, and response for a repeated key with different content. Idempotency is a data model and transaction concern, not merely a cache.

19.18 Notifications through an outbox

Sending email inside the request transaction couples database locks to an external service and creates ambiguous rollback behavior. Store an outbox event in the same transaction as the business change. A worker reads committed events, sends notifications, and records completion or retry state.

The outbox extends Chapter 18's operating-boundary principle: commit business state first, then let separately owned delivery handle retries and failure.

The worker needs bounded retries, backoff, dead-letter handling, idempotent delivery where possible, and observability.

19.19 Observability fields

Redact justification text and personal details unless a documented audit need requires them.

These fields are Chapter 12's logging guidance made concrete: correlate one request without turning sensitive payloads or unbounded identifiers into telemetry.

Field	Purpose
correlation_id	Connect events for one inbound request
request_id	Stable business entity identifier
actor_id	Who initiated the action
event	Stable event name
status_before / status_after	State transition evidence
duration_ms	Latency of owned operation
outcome	success, rejected, failed, retried

19.20 Security review checklist

The checklist is a review trigger, not proof. Each control needs implementation evidence and an owner.

- Every endpoint authenticates and authorizes the resource action.
- SQL is parameterized.
- Input sizes and collection counts are bounded.
- Secrets never appear in source, images, logs, or error responses.
- Audit events are append-only through normal application paths.
- Dependencies and container images are scanned and updated.
- Backups, restore, credential rotation, and access revocation are tested.

19.21 End-to-end walkthrough

An employee submits one monitor request. The API validates the payload, the service creates a submitted domain object, the repository stores the request and audit event, and the response returns 201 with the generated ID. Policy routes the request to automatic approval or manager review.

For manager review, an authorized manager approves. The repository updates only the expected version and stores an outbox event. The worker sends notification after commit. Metrics record latency and outcome; logs carry correlation_id and request_id through each boundary.

19.22 Capstone completion criteria

The capstone is complete when a clean environment can build and run the service, migrations create the database, automated tests cover rules and boundaries, a documented request can be submitted and reviewed, failures produce the defined error contract, and operators can identify the deployed version and diagnose a failed request.

Extensions such as inventory reservation, procurement integration, returns, attachments, and analytics should be added only after the existing transition, authorization, evidence, and recovery contracts remain intact.

- Reproducible build
- Automated migration
- Unit and integration tests
- Stable API schema
- Authorization evidence
- Audit trail
- Health and metrics
- Backup and rollback procedure

Build: complete one end-to-end request

Choose one request and follow it from transport input to stored state and emitted event. The trace should be understandable without a debugger.

111. Write the transition table including source state, command, actor, guard, destination state, and audit event.
112. Implement the domain entity and pure transition tests before routers or SQL.
113. Create repository interfaces and an in-memory implementation, then add a PostgreSQL implementation using asyncpg.
114. Implement create, view, approve, reject, comment, and fulfillment endpoints with a consistent error model.
115. Add idempotency, optimistic concurrency, audit events, and an outbox-backed notification flow in separate phases.
116. Produce a deployment and support checklist covering migrations, monitoring, backups, access review, and incident handling.

Capstone acceptance evidence

- Request trace
- Domain transition tests
- PostgreSQL integration test
- HTTP contract test
- Operational log and metric fields

Key takeaways

- Start with state: The workflow contract comes before routes.
- Separate layers: Domain rules should not depend on HTTP or PostgreSQL.
- Traceability: Comments and audit events are part of the product, not optional logging.
- Deliver in phases: Prove the core before expanding scope.

Practice

117. Add a canceled state and define who may trigger it.
118. Write tests for duplicate approval and rejection after approval.
119. Design an idempotency strategy for request submission.

Appendix A - Verification matrix

Verification reduces avoidable errors but does not prove production suitability. External services, operating systems, permissions, load, and dependency versions still require project-level tests.

Category	Verification approach
Standard-library examples	Executed under Python 3.13.5 when self-contained
Framework examples	Executed when installed; otherwise parsed or syntax-reviewed
Shell and deployment commands	Format and sequence review
SQL and configuration	Structural review; requires project environment for execution
Source working-edition examples	Original execution labels preserved

Appendix B - Corrected misconceptions

Misconception	Correction
__init__ creates the object	__new__ creates; __init__ initializes an existing instance.
Every nested function is a closure	A closure captures one or more names from an enclosing lexical scope.
ContextVar values automatically appear in all new threads	Async task propagation is built in; manually created threads require explicit context handling.
Tuple membership is O(1)	Tuple and list membership are O(n); set lookup is average O(1).
Removing __builtins__ makes eval safe	eval and exec remain arbitrary code execution mechanisms, not sandboxes.
async def makes blocking code non-blocking	Blocking libraries still block the event-loop thread.

Appendix C - Operator and collection quick reference

These are growth models, not timing guarantees. Measure representative end-to-end work.

Operation	Typical behavior
list[index]	O(1) indexed access
value in list / tuple	O(n) sequential comparison
value in set	Average O(1) hash lookup
mapping[key]	Average O(1) hash lookup
deque.appendleft / popleft	O(1) end operations
heapq push / pop	O(log n)
sorted(values)	O(n log n)

Appendix D - Exception and HTTP mapping reference

Application condition	Suggested public result	Retry?
Invalid client input	400 or 422	No, correct request
Unauthenticated	401	After authentication
Unauthorized	403	No unless permissions change
Missing resource	404	Usually no
State or version conflict	409	Reload and decide
Rate limited	429	After Retry-After
Temporary dependency failure	503	Bounded retry may be safe
Unexpected server defect	500	Client retry depends on operation idempotency

Appendix E - Packaging command reference

Portability note: shell wildcard expansion differs across PowerShell, cmd.exe, and POSIX shells. Replace <wheel-file> with the concrete wheel path produced by the build when you need a command that behaves consistently across shells.

Use a clean environment for the final wheel installation test. Upload only artifacts produced and verified by the release pipeline.

```
SHELL
python -m venv .venv
python -m pip install --upgrade pip
python -m pip install -e .
python -m pytest
python -m build
python -m pip install --force-reinstall <wheel-file>
python -m twine check dist/*
python -m twine upload dist/*
```

Appendix F - Deployment readiness checklist

- Immutable build artifact with source revision and digest
- Validated environment configuration and secret source
- Database migration and backward-compatibility plan
- Non-root runtime and minimal dependencies
- Liveness, readiness, metrics, logs, and correlation
- Resource limits, timeouts, concurrency bounds, and graceful shutdown
- Backup restore and rollback rehearsal
- Dependency, image, and access review
- Documented owner and incident path

Appendix G - Glossary: language and design

Term	Meaning
Binding	Association between a name and an object
Mutable	Object state can change without replacing the object
Iterable	Object capable of producing an iterator
Iterator	One traversal with __next__ state
Closure	Function retaining access to enclosing lexical names
Descriptor	Class attribute participating in attribute access
Protocol	Structural interface used by static typing
Invariant	Condition that valid object state must preserve

Appendix H - Glossary: production systems

Term	Meaning
Idempotency	Repeated equivalent operation has no additional effect
Backpressure	Mechanism that limits producers when consumers are saturated
Readiness	Whether an instance can safely receive traffic
Correlation ID	Identifier linking events for one request flow
Outbox	Committed event record processed asynchronously
Wheel	Built Python distribution artifact
ASGI	Asynchronous server/application interface
WSGI	Synchronous server/application interface

Appendix I - Foundation exercises

- Write a parser that preserves the difference between missing, zero, and blank input.
- Refactor a deeply nested menu into guard clauses and named functions.
- Demonstrate shallow and deep copy behavior with nested values.
- Create a package whose import performs no I/O.
- Read a UTF-8 JSON Lines file and report line-specific validation failures.
- Compare list and set membership with a reproducible benchmark.

Appendix J - Advanced exercises

- Implement a frozen Money value with validated Decimal amount and currency.
- Replace an inheritance hierarchy with Protocol-based composition and justify the boundary.
- Write sync and async timing decorators that preserve metadata.
- Build a generator pipeline and prove cleanup after a consumer failure.
- Create a bounded threaded producer-consumer system.
- Implement a TaskGroup workflow with timeout and cancellation cleanup.
- Design an outbox worker with idempotent processing.

Appendix K - Selected solution notes

Strong solutions make ownership explicit. Parsers convert once at the edge. Domain values reject impossible state. Services coordinate collaborators without importing transport frameworks. Repositories parameterize SQL and translate missing rows. Tests assert observable behavior rather than private call order.

For concurrency exercises, bound the number of in-flight operations and surface worker failures. For async exercises, remove blocking calls from the event loop and re-raise cancellation after cleanup. For deployment exercises, build once, promote the same artifact, and prove rollback compatibility.

Appendix L - Sources and verification references

Reference system. Bracketed keys in the main text point here. Primary documentation is preferred for language behavior, framework behavior, protocols, databases, packaging, deployment, and security guidance. The book's executable baseline is Python 3.13.5; later-version references are included only where behavior differs materially. URLs were rechecked on 8 August 2026. Framework and service behavior must still be verified against the versions pinned by the reader's project.

- [PY-DATA-313] Python Software Foundation. "Data model," Python 3.13 documentation. https://docs.python.org/3.13/reference/datamodel.html
- [PY-THREAD-313] Python Software Foundation. "threading - Thread-based parallelism," Python 3.13 documentation. https://docs.python.org/3.13/library/threading.html
- [PY-FREE-313] Python Software Foundation. "Python experimental support for free threading," Python 3.13 documentation. https://docs.python.org/3.13/howto/free-threading-python.html
- [PY-FREE-314] Python Software Foundation. "Python support for free threading," Python 3.14 documentation. https://docs.python.org/3.14/howto/free-threading-python.html
- [PY-THREAD-314] Python Software Foundation. "threading - Thread-based parallelism," Python 3.14 documentation; includes Thread(context=...) and build-dependent context inheritance. https://docs.python.org/3.14/library/threading.html
- [PY-CONTEXT] Python Software Foundation. "contextvars - Context Variables," Python documentation. https://docs.python.org/3/library/contextvars.html
- [PY-ASYNCIO-313] Python Software Foundation. "Coroutines and Tasks," Python 3.13 asyncio documentation; TaskGroup, cancellation, and task lifecycle. https://docs.python.org/3.13/library/asyncio-task.html
- [PY-PACKAGING] Python Packaging Authority. "Writing your pyproject.toml," Python Packaging User Guide. https://packaging.python.org/en/latest/guides/writing-pyproject-toml/
- [PY-SRC-LAYOUT] Python Packaging Authority. "src layout vs flat layout," Python Packaging User Guide. https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/
- [HTTP-9110] IETF. RFC 9110, "HTTP Semantics," especially client-error status semantics including 400 and 401. https://www.rfc-editor.org/rfc/rfc9110.html
- [FASTAPI-ASYNC] FastAPI project. "Concurrency and async / await." https://fastapi.tiangolo.com/async/
- [FASTAPI-ERRORS] FastAPI project. "Handling Errors," including request-validation error handling and overridable 422 behavior. https://fastapi.tiangolo.com/tutorial/handling-errors/
- [FLASK-ASYNC] Pallets. "Using async and await," Flask documentation; explains WSGI worker behavior for async views. https://flask.palletsprojects.com/en/stable/async-await/
- [ASYNCPG] MagicStack. "asyncpg Usage," including transactions, parameter placeholders, and connection pools. https://magicstack.github.io/asyncpg/current/usage.html
- [POSTGRES-CONCURRENCY] PostgreSQL Global Development Group. "Concurrency Control," PostgreSQL current documentation. https://www.postgresql.org/docs/current/mvcc.html
- [DOCKER-BEST] Docker. "Building best practices," including minimal bases, rebuild policy, multi-stage builds, and non-root USER guidance. https://docs.docker.com/build/building/best-practices/
- [OWASP-INPUT] OWASP. "Input Validation Cheat Sheet." https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
- [OWASP-AUTHZ] OWASP. "Authorization Cheat Sheet." https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

- [OWASP-LOGGING] OWASP. “Logging Cheat Sheet.” https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

Supplementary beginner framing sources, if used, are secondary and must not override the primary references above for technical claims. The release companion repository records the verification method and result for each code/configuration block.

Appendix M - Python foundation quick reference

This compact reference supports the foundation chapters. It is a reminder, not a replacement for running the examples and reading the surrounding design guidance.

Task	Example
Print a value	print(value)
Read text input	raw = input("Prompt: ")
Convert to integer	number = int(raw)
Check a type	isinstance(value, int)
Check absence	value is None
Loop with index	for index, item in enumerate(items):
Loop over two inputs	for left, right in zip(a, b, strict=True):
Open a text file	with open(path, "r", encoding="utf-8") as file:
Handle expected failure	except ValueError as exc:
Create a virtual environment	python -m venv .venv
Install into selected Python	python -m pip install package
Run a module	python -m package.module
Collection operation	**Preferred expression**
Add to end of list	items.append(value)
Add many values	items.extend(values)
Safe dictionary lookup	mapping.get(key, default)
Create a set	unique = set(values)
Set union	left | right
Set intersection	left & right
Sort without mutation	ordered = sorted(values)
Copy a list	copy = values.copy()
Count values	collections.Counter(values)
Queue from the left	collections.deque(values).popleft()

Appendix N - Topic navigation

End of Python from Foundations to Production. Use the quick reference as a reminder, then return to the chapter examples, labs, failure modes, and production checklists when applying the material.

Need	Go to
Environment and interpreter problems	Chapters 1 and 17
Type, identity, or mutation confusion	Chapters 2, 5, and 7
Designing replaceable behavior	Chapters 8 and 9
Large streams and cleanup	Chapter 10
Performance and algorithms	Chapters 11 and 12
Blocking I/O and CPU work	Chapter 13
High-concurrency I/O	Chapter 14
Dynamic framework mechanisms	Chapter 15
HTTP services	Chapter 16
Build and release	Chapters 17 and 18
Integrated service example	Chapter 19

Appendix O - Tools, Environments, Execution, and Debugging

Return here after Chapter 4 or whenever a project stops behaving consistently across terminals, editors, tests, and automation. These topics matter. They simply do not belong between a beginner and the first successful script.

O.1 Source, bytecode, interpreter, and effects

A .py file contains source text. CPython parses that text, compiles it to bytecode, and executes the bytecode in the Python virtual machine. Bytecode is an internal execution format, not a portable deployment artifact. A syntax error occurs before ordinary execution; imports and runtime exceptions occur later.

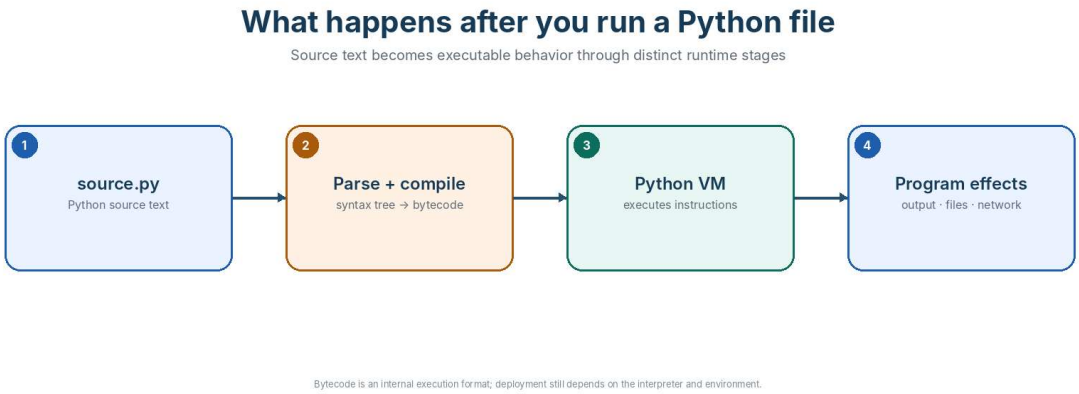

Figure O.1 - Source, bytecode, runtime, and effects are different layers.

O.2 Identify the interpreter you are actually using

Editors, terminals, test runners, and deployment systems can select different Python executables. When a package appears installed but cannot be imported, print sys.executable and sys.version before changing anything.

```
import sys
print(sys.executable)
print(sys.version)
```

O.3 Use a virtual environment per project

A virtual environment isolates a project interpreter and site-packages directory. It does not isolate operating-system libraries, network access, or secrets. Use python -m pip so pip targets the interpreter you intend to modify.

```
python -m venv .venv
# Windows PowerShell: .venv\Scripts\Activate.ps1
# macOS/Linux: source .venv/bin/activate
python -m pip install --upgrade pip
```

O.4 Configure an editor without depending on it

Select the interpreter inside .venv, then verify the same interpreter in the integrated terminal. The editor improves navigation and debugging; the terminal command remains the portable contract used by CI and production.

O.5 Expose a process exit status

Automated callers need to distinguish success from failure. A main function can return an integer, and raise SystemExit(main()) can expose that status to a shell, scheduler, or build agent.

```
def main() -> int:
    print("completed")
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

O.6 Read tracebacks as a call path

Read the final exception line first, then move upward to the first relevant frame in your code. Fix the violated assumption, not merely the line that raised. Preserve the smallest reproduction as a test.

Index

Alphabetical topic index. Page references are clickable in the digital edition. Return to Contents

V

W

PYTHON
FROM FOUNDATIONS TO PRODUCTION

Core language • design • reliability • concurrency
APIs • packaging • deployment • security

A practical learning journey from first script
to production-shaped software

Learn Python as a software-building discipline, not just a syntax tour. The book moves from scripts and functions to object design, testing, concurrency, APIs, packaging, deployment, and production judgment through one evolving workplace story.

Written by the Team of Swatantra

SWATANTRA.ai