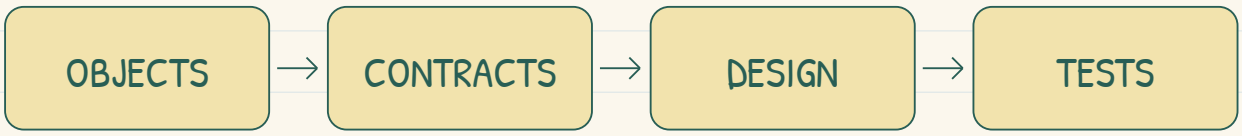
THE PRACTICAL NOTEBOOK

Object-Oriented Python

From your first object to thoughtful, testable design.

80 pages / 24 guided lessons / 48 code files



Understand it. Try it. Explain it.

A patient, practical route from basic Python to advanced object-oriented design. Learn with small examples, predict their behaviour, solve exercises and build a tested library-lending project. No paid tools, accounts or third-party packages are needed for the practice files.

PRIVATUS COSMAS / PRIVER

depriver.tech · First edition / October 2026

Your 24-lesson route

Work in order, or use this clickable map to revisit a topic. Each lesson has a learn page, a practice page and an explained solution.

01	Start with ordinary Python	4
02	Classes, objects and self	7
03	State and useful methods	10
04	Instance data and shared data	13
05	Encapsulation and properties	16
06	Factories and method types	19
07	Composition: objects together	22
08	Inheritance and substitution	25
09	Polymorphism and duck typing	28
10	ABCs and structural protocols	31
11	Representations and equality	34
12	Data classes and value objects	37
13	Collections and iteration	40
14	Context managers and cleanup	43
15	Errors and atomic state changes	46
16	Test behaviour, not internals	49
17	Dependencies and test doubles	52
18	Design principles and strategies	55
19	Multiple inheritance and MRO	58
20	Descriptors and attribute lookup	61
21	Construction hooks and metaclasses	64
22	Memory, copying and slots	67
23	Capstone: a lending domain	70
24	Capstone: boundaries and review	73

Setup and study method · 3 References and study aids · 76-80

Make this book yours

BEFORE YOU START

No OOP knowledge is assumed. Lesson 1 refreshes names, functions and lists. If programming itself is new, first work through Python: the practical notebook (depriver.tech/archive/python-the-practical-notebook.pdf), especially variables, loops, functions and tests. Then return here.

YOUR WORKSPACE

Install Python 3.11 or newer. Open a terminal in the extracted practice folder; on Windows, use py instead of python if needed. Use a text editor and save files with a .py extension. Examples use only the standard library. Tested versions are recorded in verification.json.

TERMINAL / FOLLOW THE COMMANDS FOR THIS BOOK

```
python --version
python lesson_01_example.py
python lesson_01_solution.py
python -m unittest discover -s . -v
```

A SIMPLE STUDY RHYTHM

1. Read: predict what the example will do before running it.
2. Practise: write your answer using the acceptance cases.
3. Compare: read the solution, explain the difference, then extend it.

The companion ZIP contains lesson_01_example.py through lesson_24_example.py and matching solution files. Run them from the extracted folder; lesson 24 uses capstone.py. Most successful checks are silent; a traceback means a check failed.

READABLE NOTES, PRECISE CODE

Handwritten headings keep the notebook feel; body text and code use clear fonts. Copy runnable code from the ZIP to preserve indentation. The PDF is searchable and its contents links are clickable.

Start with ordinary Python

THE GOAL

Read a short function before writing your first class.

A program stores information and performs operations on it. A name refers to a value; a list holds several items. A function is a named piece of reusable behaviour. Indentation groups its instructions. In this example, a dictionary holds one learner and the function builds a greeting. OOP will bring related information and behaviour together.

lesson_01_example.py

```
def greet(learner):
    return "Hello, " + learner["name"]

learner = {"name": "Asha", "topics": []}
learner["topics"].append("Python")
assert greet(learner) == "Hello, Asha"
assert len(learner["topics"]) == 1
print(greet(learner))
```

The function receives the dictionary as its parameter. return sends a result back; print displays it. assert checks a prediction and raises AssertionError if it is false. Expected output: Hello, Asha. A line beginning with # is a comment for readers.

WATCH FOR THIS

Use four spaces per indentation level. Read the final traceback line first when an example fails; check spelling and indentation before changing the design.

Your turn: Start with ordinary Python

THE CHALLENGE

Write completed_count(learner). Count the topics in the learner dictionary and return that number. Do not print inside the function.

ACCEPTANCE CHECKS

[ ] No topics gives 0.

[ ] Two topics gives 2.

[ ] Calling the function does not change the dictionary.

A NUDGE, IF YOU NEED ONE

Use len on the list stored under the topics key.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

START WITH ORDINARY PYTHON

lesson_01_solution.py

```
def completed_count(learner):
    return len(learner["topics"])

empty = {"name": "Asha", "topics": []}
ready = {"name": "Juma", "topics": ["Git", "Python"]}
assert completed_count(empty) == 0
assert completed_count(ready) == 2
assert ready["topics"] == ["Git", "Python"]
```

WHY IT WORKS

The function reads the list without changing it. Returning the number allows another function to use it. This is a useful design even without a class: OOP is one way to organise a program, not a requirement for every task.

TAKE IT FURTHER

Practise loops and functions in Python: the practical notebook. Write a loop that greets three learners before continuing.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Classes, objects and self

THE GOAL

Create two independent objects from one class.

Imagine a library catalogue: every book has a title, but different books have different titles. A class describes a kind of object. Calling it creates an instance. An attribute stores information on that instance. A method is behaviour accessed through it. self is the conventional name for the instance receiving a method call.

lesson_02_example.py

```
class Book:
    def __init__(self, title):
        self.title = title

    def label(self):
        return "Book: " + self.title

first = Book("Python")
second = Book("Git")
assert first.label() == "Book: Python"
assert second.title == "Git"
assert first is not second
```

Book("Python") initialises first with its own title. first.label() supplies first as self automatically; Book.label(first) is equivalent here. __init__ prepares an already-created instance and must not return a replacement object.

WATCH FOR THIS

Forgetting self in a method signature causes argument-count errors. Merely writing name = name in __init__ does not create an instance attribute.

Your turn: Classes, objects and self

THE CHALLENGE

Create a Learner with a name and introduce() method. Two learners must introduce themselves using their own names.

ACCEPTANCE CHECKS

[ ] Asha introduces herself as "I am Asha".

[ ] Juma introduces himself as "I am Juma".

[ ] Changing one name does not change the other.

A NUDGE, IF YOU NEED ONE

Use self.name both when storing and when reading the name.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CLASSES, OBJECTS AND SELF

lesson_02_solution.py

```
class Learner:
    def __init__(self, name):
        self.name = name

    def introduce(self):
        return "I am " + self.name

a = Learner("Asha")
b = Learner("Juma")
assert a.introduce() == "I am Asha"
assert b.introduce() == "I am Juma"
a.name = "Neema"
assert b.name == "Juma"
```

WHY IT WORKS

The class supplies shared behaviour, while each object carries its own state. self is an ordinary parameter name, but following the convention makes code understandable to other Python programmers.

TAKE IT FURTHER

Draw two boxes for the objects and write their attributes inside. Explain the difference between Book, Book("Python") and first.title.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

State and useful methods

THE GOAL

Model behaviour that preserves a simple rule.

State is what an object remembers now. A method can change that state. A useful class makes valid operations easy to express: a counter advances through increment(), rather than requiring every caller to know its storage details. Start with one responsibility and write down the rule that should remain true after every public operation.

lesson_03_example.py

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1
        return self.value

counter = Counter()
assert counter.increment() == 1
assert counter.increment() == 2
```

The same instance remembers the previous call. A new Counter starts at zero. A method that only reads is a query; a method that changes state is a command. Some APIs return a useful result from a command, but document that choice.

WATCH FOR THIS

A public attribute remains writable. A method-level validation rule does not stop a caller from assigning p.pages = -1. We will refine the API with properties.

Your turn: State and useful methods

THE CHALLENGE

Build a ReadingProgress object starting at zero pages. advance(pages) accepts a positive integer and returns the new total. Reject zero, negatives and non-integers.

ACCEPTANCE CHECKS

[ ] advance(3), then advance(2), gives 5.

[ ] advance(0) raises ValueError.

[ ] A failed call leaves the total unchanged.

A NUDGE, IF YOU NEED ONE

Validate before updating state. bool is a subclass of int; decide whether True should count as a page.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

STATE AND USEFUL METHODS

lesson_03_solution.py

```
class ReadingProgress:
    def __init__(self):
        self.pages = 0

    def advance(self, pages):
        if type(pages) is not int or pages <= 0:
            raise ValueError("positive integer required")
        self.pages += pages
        return self.pages

p = ReadingProgress()
assert p.advance(3) == 3
assert p.advance(2) == 5
try:
    p.advance(0)
except ValueError:
    pass
else:
    raise AssertionError("zero accepted")
assert p.pages == 5
```

WHY IT WORKS

Validation happens before mutation, so rejected operations preserve the previous state. type(pages) is int deliberately excludes booleans and integer subclasses in this small contract.

TAKE IT FURTHER

Test -1, True and "3". Add a reset() command and describe whether resetting should return a value.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Instance data and shared data

THE GOAL

Prevent accidental sharing between objects.

A class attribute belongs to the class and is normally found when an instance has no overriding attribute. It is useful for a shared constant. Mutable per-object data belongs in __init__. A list declared in the class body is one shared list, which can make unrelated objects change together.

lesson_04_example.py

```
class Shelf:
    category = "learning"

    def __init__(self):
        self.titles = []

left = Shelf()
right = Shelf()
left.titles.append("Python")
assert right.titles == []
assert left.category == right.category == "learning"
```

Each __init__ call evaluates [] again. Both shelves see the class category, while titles is stored separately on each shelf. Setting left.category would usually shadow the class attribute for left; mutating a shared list would not make a new list.

WATCH FOR THIS

A shallow copy still shares nested mutable objects. See Python with DSA, Lists and copying, for a related exercise.

Your turn: Instance data and shared data

THE CHALLENGE

Create a StudyGroup with a fresh members list. Accept optional initial members and copy that outer list so later caller appends do not affect the group.

ACCEPTANCE CHECKS

[ ] Two empty groups do not share members.

[ ] Changing the input list does not change the group.

[ ] add(name) adds one member.

A NUDGE, IF YOU NEED ONE

Use None as the default argument and list(members) for an outer copy.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

INSTANCE DATA AND SHARED DATA

lesson_04_solution.py

```
class StudyGroup:
    def __init__(self, members=None):
        self.members = [] if members is None else list(members)

    def add(self, name):
        self.members.append(name)

source = ["Asha"]
a = StudyGroup(source)
b = StudyGroup()
source.append("Juma")
a.add("Neema")
assert a.members == ["Asha", "Neema"]
assert b.members == []
```

WHY IT WORKS

A default argument is evaluated when the function is defined, not for each call. None avoids a persistent default list. Copying the supplied list makes ownership of the outer container explicit.

TAKE IT FURTHER

Put dictionaries inside source and mutate one. Explain why copying only the outer list does not isolate its contents.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Encapsulation and properties

THE GOAL

Expose a small API that protects an invariant.

Encapsulation means giving callers a coherent public interface while keeping implementation decisions inside. An invariant is a condition valid state must satisfy. A property lets attribute access run controlled behaviour. A leading underscore signals internal use; it is a convention, not an access-control barrier.

lesson_05_example.py

```
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("below absolute zero")
        self._celsius = value

t = Temperature(20)
t.celsius = 25
assert t.celsius == 25
```

The @ lines apply decorators to the functions below them. The getter reads _celsius; the setter validates assignments. Construction uses the same setter. This introductory model assumes ordinary finite numeric inputs; a public scientific API would validate types and non-finite numbers too.

WATCH FOR THIS

Double underscores trigger name mangling to reduce subclass name collisions; they do not make secrets secure. Avoid Java-style getters and setters when ordinary attributes are sufficient.

Your turn: Encapsulation and properties

THE CHALLENGE

Create a Stock object with a read-only quantity property. remove(amount) accepts positive integers up to the available quantity. Reject invalid construction and invalid removals.

ACCEPTANCE CHECKS

[ ] Stock(3).remove(2) leaves 1.

[ ] Removing too much raises ValueError without mutation.

[ ] Negative starting stock is rejected.

A NUDGE, IF YOU NEED ONE

Store _quantity internally and expose only a property getter.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ENCAPSULATION AND PROPERTIES

lesson_05_solution.py

```
class Stock:
    def __init__(self, quantity):
        if type(quantity) is not int or quantity < 0:
            raise ValueError("invalid quantity")
        self._quantity = quantity

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

    def remove(self, amount):
        if type(amount) is not int or amount <= 0:
            raise ValueError("invalid amount")
        if amount > self._quantity:
            raise ValueError("insufficient stock")
        self._quantity -= amount

stock = Stock(3)
stock.remove(2)
assert stock.quantity == 1
```

WHY IT WORKS

The public quantity attribute has no setter, so normal assignment to it raises AttributeError. Callers request a meaningful operation instead of writing an arbitrary new total. Internal _quantity is still accessible to a determined caller.

TAKE IT FURTHER

Write rejection tests for construction, removal and quantity assignment. Explain which checks protect an invariant and which merely document an internal convention.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Factories and method types

THE GOAL

Choose instance, class and static methods deliberately.

An instance method works with self. A class method receives cls, the class used to call it, and often provides an alternative constructor. A static method receives no implicit object or class. It is a namespaced function; use a module-level function when the class adds no useful meaning.

lesson_06_example.py

```
class Book:
    def __init__(self, title):
        self.title = title

    @classmethod
    def from_line(cls, line):
        return cls(line.strip())

    @staticmethod
    def valid_title(title):
        return bool(title.strip())

assert Book.from_line(" Python ").title == "Python"
assert not Book.valid_title("   ")
```

from_line uses cls instead of hard-coding Book. A compatible subclass inherits a factory that constructs that subclass. valid_title needs no stored state. Validation helpers do not enforce a rule unless the construction or mutation path actually calls them.

WATCH FOR THIS

A subclass that changes its constructor contract may break an inherited factory. Inheritance requires compatible expectations, not merely matching method names.

Your turn: Factories and method types

THE CHALLENGE

Create a Coordinate(x, y) with from_text("3,4"). Parse exactly two integers. Return an instance of cls; let malformed numeric input raise ValueError.

ACCEPTANCE CHECKS

[ ] "3,4" becomes x=3, y=4.

[ ] Whitespace around numbers is accepted.

[ ] Three coordinates and non-numeric input fail.

A NUDGE, IF YOU NEED ONE

Unpack text.split(",") into exactly two names, then convert with int.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

FACTORIES AND METHOD TYPES

lesson_06_solution.py

```
class Coordinate:
    def __init__(self, x, y):
        self.x, self.y = x, y

    @classmethod
    def from_text(cls, text):
        x, y = text.split(",")
        return cls(int(x), int(y))

point = Coordinate.from_text("3, 4")
assert (point.x, point.y) == (3, 4)
class ScreenPoint(Coordinate):
    pass
assert isinstance(ScreenPoint.from_text("1,2"), ScreenPoint)
```

WHY IT WORKS

Unpacking enforces exactly two components; integer conversion enforces the numeric format. The factory translates a representation into an object. Keeping parsing outside __init__ also allows direct construction from already-parsed numbers.

TAKE IT FURTHER

Add from_pair((x, y)). Compare these alternative constructors with one constructor containing several unrelated input-format branches.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Composition: objects together

THE GOAL

Build behaviour by delegating to another object.

Composition gives one object a reference to another collaborator. A reading service has a catalogue; it is not a catalogue. The service can delegate lookup while keeping its own responsibility small. Injecting the collaborator through the constructor makes that dependency visible and replaceable in a test.

lesson_07_example.py

```
class Catalogue:
    def title_for(self, code):
        return {"P1": "Python"}[code]

class ReadingDesk:
    def __init__(self, catalogue):
        self.catalogue = catalogue

    def recommend(self, code):
        return "Read " + self.catalogue.title_for(code)

desk = ReadingDesk(Catalogue())
assert desk.recommend("P1") == "Read Python"
```

ReadingDesk knows the operation it needs, title_for. It does not need to know whether the collaborator uses a dictionary or a database. This is a small version of the boundary design in Software Architecture: the practical fieldbook.

WATCH FOR THIS

Do not add inheritance merely because two classes share a few lines. A function or a composed collaborator may express the relationship more clearly.

Your turn: Composition: objects together

THE CHALLENGE

Build a Checkout that receives a price policy. total(quantity) delegates to policy.price(quantity). Implement a fixed policy costing 500 whole TZS units per item.

ACCEPTANCE CHECKS

[ ] Quantity 0 costs 0.

[ ] Quantity 3 costs 1500.

[ ] A second policy can be supplied without editing Checkout.

A NUDGE, IF YOU NEED ONE

The policy is an object supplied to Checkout.__init__; do not create it inside total.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

COMPOSITION: OBJECTS TOGETHER

lesson_07_solution.py

```
class FixedPrice:
    def price(self, quantity):
        return quantity * 500

class Checkout:
    def __init__(self, policy):
        self.policy = policy

    def total(self, quantity):
        return self.policy.price(quantity)

checkout = Checkout(FixedPrice())
assert checkout.total(0) == 0
assert checkout.total(3) == 1500
```

WHY IT WORKS

The dependency is explicit and the policy owns the pricing rule. Values are whole units in an educational example. The contract here assumes a non-negative integer quantity; a public boundary must enforce that assumption.

TAKE IT FURTHER

Create a second policy with a quantity discount. Keep checkout unchanged and test both policies at the discount boundary.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Inheritance and substitution

THE GOAL

Use a subclass when it honours its parent contract.

Inheritance makes a specialised type reuse or refine behaviour. A useful subclass can stand in for its base class without surprising callers. Overriding replaces a method implementation. super() delegates through the method resolution order; in this simple example it reaches the base implementation.

lesson_08_example.py

```
class Notice:
    def render(self):
        return "Library closes at 17:00"

class UrgentNotice(Notice):
    def render(self):
        return "URGENT: " + super().render()

def display(notice):
    return notice.render()

assert display(UrgentNotice()).startswith("URGENT:")
assert display(Notice()) == "Library closes at 17:00"
```

Both types satisfy the same small contract: render() returns text. The caller does not need a type switch. If a subclass unexpectedly returned a list or refused normal inputs accepted by the base, substitution would become unreliable.

WATCH FOR THIS

An is-a sentence alone does not prove correct inheritance. Check accepted inputs, result meaning, exceptions and preserved invariants.

Your turn: Inheritance and substitution

THE CHALLENGE

Create a Report with heading() returning "Report". Create StudyReport that calls the base constructor to keep a title and overrides heading() to include that title.

ACCEPTANCE CHECKS

[ ] Report("Week 1").heading() returns "Report".

[ ] StudyReport("Week 1").heading() returns "Study: Week 1".

[ ] Both objects retain their title.

A NUDGE, IF YOU NEED ONE

If the subclass has no extra initialisation, it can inherit __init__ directly.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

INHERITANCE AND SUBSTITUTION

lesson_08_solution.py

```
class Report:
    def __init__(self, title):
        self.title = title

    def heading(self):
        return "Report"

class StudyReport(Report):
    def heading(self):
        return "Study: " + self.title

base = Report("Week 1")
report = StudyReport("Week 1")
assert base.heading() == "Report"
assert report.heading() == "Study: Week 1"
assert report.title == base.title
```

WHY IT WORKS

StudyReport inherits the constructor automatically. There is no reason to duplicate it merely to call super().__init__. Add an override only when the subclass genuinely needs a different operation.

TAKE IT FURTHER

Design an invalid subclass that returns None from heading(). Explain which caller expectation it breaks and repair the design with composition if appropriate.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Polymorphism and duck typing

THE GOAL

Call a shared operation on different implementations.

Polymorphism lets the same caller work with different implementations. Python often uses duck typing: call the behaviour you need instead of demanding a particular ancestry. A shared method name is only the beginning; agree on parameters, results and failure behaviour as well.

lesson_09_example.py

```
class Plain:
    def format(self, title):
        return title

class Bracketed:
    def format(self, title):
        return "[" + title + "]"

def label(formatter, title):
    return formatter.format(title)

assert label(Plain(), "Python") == "Python"
assert label(Bracketed(), "Python") == "[Python]"
```

Neither formatter inherits from the other. The label function depends on the format(title) contract. Calling the wrong object will fail at runtime; later, Protocol can describe this contract to a static type checker.

WATCH FOR THIS

Duck typing does not mean ignoring contracts. Two send methods can have incompatible semantics even when their signatures match.

Your turn: Polymorphism and duck typing

THE CHALLENGE

Write notify(sender, text) that calls send(text). Supply a RecordingSender which stores sent texts and an UpperSender which stores their uppercase versions.

ACCEPTANCE CHECKS

[ ] RecordingSender preserves the original text.

[ ] UpperSender stores uppercase text.

[ ] Each sender owns a separate messages list.

A NUDGE, IF YOU NEED ONE

Keep the caller independent of implementation names; do not branch on isinstance.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

POLYMORPHISM AND DUCK TYPING

lesson_09_solution.py

```
class RecordingSender:
    def __init__(self):
        self.messages = []

    def send(self, text):
        self.messages.append(text)

class UpperSender(RecordingSender):
    def send(self, text):
        self.messages.append(text.upper())

def notify(sender, text):
    sender.send(text)

a, b = RecordingSender(), UpperSender()
notify(a, "hello")
notify(b, "hello")
assert a.messages == ["hello"]
assert b.messages == ["HELLO"]
```

WHY IT WORKS

The caller only requests delivery. Both implementations record locally; neither sends a real message. UpperSender is suitable only where the sender contract permits formatting changes. If exact preservation is promised, it would violate that contract.

TAKE IT FURTHER

Rewrite UpperSender as an independent class. Then implement it as a wrapper around another sender and compare the dependencies.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

ABCs and structural protocols

THE GOAL

Make an interface explicit when it adds clarity.

An abstract base class (ABC) can prevent construction until required abstract methods are implemented. A Protocol expresses a structural interface for static checking: an object can fit without explicitly inheriting it. Choose an ABC for a controlled hierarchy, or a Protocol for collaboration across unrelated types.

lesson_10_example.py

```
from abc import ABC, abstractmethod

class Exporter(ABC):
    @abstractmethod
    def export(self, title):
        raise NotImplementedError

class TextExporter(Exporter):
    def export(self, title):
        return title + "\n"

assert TextExporter().export("Python") == "Python\n"
```

An abstract method marks a required operation; simply raising NotImplementedError without ABC machinery does not block construction. Neither mechanism proves the implementation has the intended semantics. Tests still matter.

WATCH FOR THIS

@runtime_checkable permits limited runtime structural checks; it does not verify full signatures or business rules. Do not treat it as input validation.

Your turn: ABCs and structural protocols

THE CHALLENGE

Define a Formatter Protocol with format(title: str) -> str. Annotate make_label and use an unrelated class that satisfies the interface.

ACCEPTANCE CHECKS

[ ] The implementation need not inherit Formatter.

[ ] make_label returns the expected label.

[ ] Explain why annotations alone do not validate runtime inputs.

A NUDGE, IF YOU NEED ONE

Import Protocol from typing and put an ellipsis in the protocol method body.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ABCS AND STRUCTURAL PROTOCOLS

lesson_10_solution.py

```
from typing import Protocol

class Formatter(Protocol):
    def format(self, title: str) -> str: ...

class PrefixFormatter:
    def format(self, title: str) -> str:
        return "Read: " + title

def make_label(formatter: Formatter, title: str) -> str:
    return formatter.format(title)

assert make_label(PrefixFormatter(), "Git") == "Read: Git"
```

WHY IT WORKS

A static checker can compare the implementation signature with the protocol. The interpreter still executes normal method calls. This lesson checks runtime behaviour; it does not claim that a separate type checker has been run.

TAKE IT FURTHER

Read Python for Backend, Dependencies and test doubles. Describe a repository Protocol for looking up a book by code without exposing database details.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Representations and equality

THE GOAL

Give objects clear displays and meaningful comparisons.

__repr__ is a diagnostic representation; __str__ is a user-facing display. Equality asks whether two objects mean the same value, while is asks whether they are the same object. Define equality from the domain contract. For unsupported operand types, return NotImplemented so Python can try the other operand appropriately.

lesson_11_example.py

```
class BookCode:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f"BookCode({self.value!r})"

    def __eq__(self, other):
        if type(other) is not type(self):
            return NotImplemented
        return self.value == other.value

a, b = BookCode("P1"), BookCode("P1")
assert a == b and a is not b
assert repr(a) == "BookCode('P1')"
```

The !r formatting uses the field representation, preserving quotes around strings. The example uses exact-type equality to avoid inventing cross-type semantics. Defining __eq__ without __hash__ makes this ordinary mutable class unhashable.

WATCH FOR THIS

Never include passwords or tokens in repr. Do not use is to compare strings or numbers by value, even if an interpreter sometimes reuses those objects.

Your turn: Representations and equality

THE CHALLENGE

Create a Label value object with diagnostic repr and user-facing str. Two Labels compare by their text. Do not add hashing to this mutable object.

ACCEPTANCE CHECKS

[ ] str(Label("Python")) is "Python".

[ ] repr includes the class name and quoted value.

[ ] Two equal texts compare equal; comparison with a string is false.

A NUDGE, IF YOU NEED ONE

Return NotImplemented for other types, not an exception.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

REPRESENTATIONS AND EQUALITY

lesson_11_solution.py

```
class Label:
    def __init__(self, text):
        self.text = text

    def __str__(self):
        return self.text

    def __repr__(self):
        return f"Label({self.text!r})"

    def __eq__(self, other):
        if type(other) is not type(self):
            return NotImplemented
        return self.text == other.text

assert str(Label("Python")) == "Python"
assert repr(Label("Python")) == "Label('Python')"
assert Label("Python") == Label("Python")
assert Label("Python") != "Python"
```

WHY IT WORKS

Representation and equality serve different audiences. A mutable object can support value equality without being a valid dictionary key. Equal hashable objects must have equal hashes, and their hash must stay stable while used as keys.

TAKE IT FURTHER

Attempt to place Label in a set and explain the TypeError. Revisit Python with DSA, Hash maps and counting, for the role of keys.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Data classes and value objects

THE GOAL

Use generated methods without losing domain rules.

A data class reduces repeated initialisation and representation code for records. Type annotations declare fields but do not validate their runtime values. __post_init__ can enforce domain rules. frozen=True blocks ordinary field reassignment; it does not recursively freeze nested containers.

lesson_12_example.py

```
from dataclasses import dataclass

@dataclass(frozen=True)
class BookCode:
    value: str

    def __post_init__(self):
        if not self.value.strip():
            raise ValueError("empty book code")

assert BookCode("P1") == BookCode("P1")
assert len({BookCode("P1"), BookCode("P1")}) == 1
```

The generated equality compares field values for matching classes. This frozen object holds a string, so its generated hash is suitable for a set. The example assumes a str input; annotations alone would not stop an integer reaching __post_init__.

WATCH FOR THIS

frozen=True is not a security boundary. A frozen object containing a list still allows mutation of that list and may be unhashable.

Your turn: Data classes and value objects

THE CHALLENGE

Create a mutable ReadingList data class with owner and titles. Use a separate default list for each instance and test that adding to one does not affect another.

ACCEPTANCE CHECKS

[ ] Two lists with the same fields compare equal.

[ ] Default titles lists are separate objects.

[ ] Appending to one list leaves the other empty.

A NUDGE, IF YOU NEED ONE

Use field(default_factory=list), which calls list for each new instance.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DATA CLASSES AND VALUE OBJECTS

lesson_12_solution.py

```
from dataclasses import dataclass, field

@dataclass
class ReadingList:
    owner: str
    titles: list[str] = field(default_factory=list)

a = ReadingList("Asha")
b = ReadingList("Asha")
assert a == b
assert a.titles is not b.titles
a.titles.append("Python")
assert b.titles == []
assert a != b
```

WHY IT WORKS

The factory solves the mutable-default problem at object construction. Generated equality reflects the current fields, so mutation can change whether two objects compare equal. A record with several meaningful behaviours can still be a data class.

TAKE IT FURTHER

Model an immutable BookRecord with a tuple of tags. Explain whether two catalogue entries with the same title but different codes should be equal.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Collections and iteration

THE GOAL

Make a domain collection work with ordinary Python.

A custom collection should cooperate with familiar operations when those operations have an obvious meaning. __len__ supports len, __iter__ supplies an iterator for a loop, and __contains__ can define membership. Prefer wrapping a built-in collection to reimplementing its storage machinery.

lesson_13_example.py

```
class Shelf:
    def __init__(self, titles):
        self._titles = tuple(titles)

    def __len__(self):
        return len(self._titles)

    def __iter__(self):
        return iter(self._titles)

    def __contains__(self, title):
        return title in self._titles

shelf = Shelf(["Git", "Python"])
assert len(shelf) == 2
assert list(shelf) == ["Git", "Python"]
assert "Git" in shelf
```

Every call to __iter__ returns a fresh iterator, so the shelf can be traversed again. An iterator is a stateful traversal object with __next__; an iterable supplies iterators. Keeping these roles separate prevents a second loop from unexpectedly seeing no items.

WATCH FOR THIS

Do not return self from __iter__ unless the object intentionally represents a single advancing iterator. A reusable container normally returns a separate iterator.

Your turn: Collections and iteration

THE CHALLENGE

Create Countdown(start), an iterable that yields start down to 1. Reject negative or non-integer start values. Repeated iteration must produce the same sequence.

ACCEPTANCE CHECKS

[ ] Countdown(3) yields 3, 2, 1.

[ ] Countdown(0) yields no items.

[ ] Two calls to list on the same object agree.

A NUDGE, IF YOU NEED ONE

Use a generator method: yield values from range(start, 0, -1).

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

COLLECTIONS AND ITERATION

lesson_13_solution.py

```
class Countdown:
    def __init__(self, start):
        if type(start) is not int or start < 0:
            raise ValueError("non-negative integer required")
        self.start = start

    def __iter__(self):
        for value in range(self.start, 0, -1):
            yield value

countdown = Countdown(3)
assert list(countdown) == [3, 2, 1]
assert list(countdown) == [3, 2, 1]
assert list(Countdown(0)) == []
```

WHY IT WORKS

Calling the generator method creates a new generator each time. yield pauses one traversal; return ends it. A manual __next__ implementation would need to raise StopIteration when exhausted.

TAKE IT FURTHER

Add integer indexing to Shelf using __getitem__. Decide whether slicing returns a tuple or another Shelf and document that choice.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Context managers and cleanup

THE GOAL

Guarantee cleanup when a block succeeds or fails.

A context manager defines entry and exit around a with block. __enter__ returns the value bound after as. If entry succeeds, __exit__ runs when the block leaves, including through an exception. This makes resource lifetime explicit instead of depending on when garbage collection happens.

lesson_14_example.py

```
class Session:
    def __init__(self):
        self.closed = False

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc, traceback):
        self.closed = True
        return False

with Session() as session:
    assert not session.closed
assert session.closed
```

This session is an educational state model, not a network connection. Returning False from __exit__ allows an exception to propagate. Returning True suppresses it, which should be an explicit and narrow policy.

WATCH FOR THIS

__del__ is not a reliable place for timely cleanup. Use with for files and other resources; avoid hiding unrelated errors by always returning True.

Your turn: Context managers and cleanup

THE CHALLENGE

Implement RecordingScope that records "enter" and "exit" in a supplied list. Verify exit still runs when the block raises ValueError and that the exception reaches the caller.

ACCEPTANCE CHECKS

[ ] A normal block records enter then exit.

[ ] An exceptional block still records exit.

[ ] The original ValueError is not suppressed.

A NUDGE, IF YOU NEED ONE

Wrap the with statement in try/except, not only the statement inside its block.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CONTEXT MANAGERS AND CLEANUP

lesson_14_solution.py

```
class RecordingScope:
    def __init__(self, events):
        self.events = events

    def __enter__(self):
        self.events.append("enter")
        return self

    def __exit__(self, *error):
        self.events.append("exit")
        return False

events = []
try:
    with RecordingScope(events):
        raise ValueError("example")
except ValueError:
    pass
else:
    raise AssertionError("error suppressed")
assert events == ["enter", "exit"]
```

WHY IT WORKS

The outer handler receives the exception after cleanup. If __enter__ itself fails, Python does not call this context manager's __exit__; partial acquisition must be handled inside entry or with an appropriate helper.

TAKE IT FURTHER

Read Python for Automation on file operations. Replace a manually opened file with with open(...), then explain the analogous __aenter__/__aexit__ roles for async with.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Errors and atomic state changes

THE GOAL

Leave an object valid when an operation is rejected.

Exceptions signal that an operation could not fulfil its contract. Validate prerequisites before mutation when possible. A domain exception lets callers distinguish an expected business rejection from a programming mistake. Catch only errors you can meaningfully handle; a blanket except can conceal a broken object.

lesson_15_example.py

```
class UnavailableError(Exception):
    pass

class Copy:
    def __init__(self):
        self.borrower = None

    def lend(self, borrower):
        if self.borrower is not None:
            raise UnavailableError("already on loan")
        self.borrower = borrower

copy = Copy()
copy.lend("Asha")
assert copy.borrower == "Asha"
```

The state transition is available to borrowed. A second loan is rejected before replacing the borrower. This is only a single-process object model; protecting a database record from concurrent requests requires a transaction or another concurrency strategy.

WATCH FOR THIS

Validating before mutation is not the same as providing a database transaction. Do not claim a multi-object operation is atomic merely because each individual method validates.

Your turn: Errors and atomic state changes

THE CHALLENGE

Create an Account of integer learning credits. spend(amount) must reject non-positive amounts and insufficient funds without altering balance. Use a custom InsufficientCredits exception.

ACCEPTANCE CHECKS

[ ] Spending 3 from 5 leaves 2.

[ ] Spending 6 from 5 raises InsufficientCredits.

[ ] Invalid amounts raise ValueError.

A NUDGE, IF YOU NEED ONE

Separate invalid input from a valid request that cannot be fulfilled.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

ERRORS AND ATOMIC STATE CHANGES

lesson_15_solution.py

```
class InsufficientCredits(Exception):
    pass

class Account:
    def __init__(self, balance):
        if type(balance) is not int or balance < 0:
            raise ValueError("invalid balance")
        self.balance = balance

    def spend(self, amount):
        if type(amount) is not int or amount <= 0:
            raise ValueError("invalid amount")
        if amount > self.balance:
            raise InsufficientCredits("not enough credits")
        self.balance -= amount

account = Account(5)
account.spend(3)
assert account.balance == 2
```

WHY IT WORKS

Input validation and domain rejection have different meanings, so their exception types differ. All checks precede the assignment. The example uses learning credits, not a complete financial ledger.

TAKE IT FURTHER

Write a failed-spend test that checks balance afterwards. Sketch how a transfer between two accounts could fail halfway and which layer should own the transaction.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Test behaviour, not internals

THE GOAL

Use unittest to protect public promises.

A useful test describes what a caller can observe: a result, an expected exception, or a state transition through a public API. Each test should arrange its own objects so test order does not matter. Standard-library unittest provides discovery and assertions without an extra dependency.

lesson_16_example.py

```
import unittest

class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

class CounterTests(unittest.TestCase):
    def test_increment(self):
        counter = Counter()
        counter.increment()
        self.assertEqual(counter.value, 1)

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

Running the file executes one test and reports OK. unittest assertions remain active with Python optimisation; plain assert statements in other examples should be run without -O. A test name should describe a behaviour that matters.

WATCH FOR THIS

A passing happy-path test does not establish correctness at boundaries. Include rejected input, repeated operations, empty collections and independent instances where relevant.

Your turn: Test behaviour, not internals

THE CHALLENGE

Implement a Gate that allows entry only when open. Test rejection while closed and success after open(). Use assertRaises for the expected failure.

ACCEPTANCE CHECKS

[ ] A new gate rejects entry.

[ ] An open gate returns "welcome".

[ ] Each test creates its own gate.

A NUDGE, IF YOU NEED ONE

Use with self.assertRaises(...) around the single failing call.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TEST BEHAVIOUR, NOT INTERNALS

lesson_16_solution.py

```
import unittest

class Gate:
    def __init__(self):
        self.is_open = False
    def open(self):
        self.is_open = True
    def enter(self):
        if not self.is_open:
            raise ValueError("closed")
        return "welcome"

class GateTests(unittest.TestCase):
    def test_closed(self):
        with self.assertRaises(ValueError):
            Gate().enter()
    def test_open(self):
        gate = Gate()
        gate.open()
        self.assertEqual(gate.enter(), "welcome")

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

WHY IT WORKS

The tests exercise the public behaviour instead of comparing private dictionaries or inspecting source code. The failure-path test would fail if Gate silently allowed entry while closed.

TAKE IT FURTHER

Use Git: the practical notebook to commit a passing version, introduce a deliberate defect, observe a test fail and restore the implementation.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Dependencies and test doubles

THE GOAL

Replace external effects with small explicit collaborators.

A dependency is something an operation needs to do its work. Constructor injection passes that dependency into the object. A fake is a simplified working implementation used in tests. It should preserve the relevant contract while avoiding actual email, networks or databases.

lesson_17_example.py

```
class ReminderService:
    def __init__(self, sender):
        self.sender = sender

    def remind(self, name):
        self.sender.send("Return your book, " + name)

class FakeSender:
    def __init__(self):
        self.messages = []
    def send(self, message):
        self.messages.append(message)

sender = FakeSender()
ReminderService(sender).remind("Asha")
assert sender.messages == ["Return your book, Asha"]
```

The service creates the message; the sender owns delivery. The test can inspect the fake through its simple recording interface. This checks the service logic, not whether a real mail provider accepts a message.

WATCH FOR THIS

A fake that behaves unlike the real adapter can give false confidence. Keep contract tests for adapters and integration tests for actual external boundaries.

Your turn: Dependencies and test doubles

THE CHALLENGE

Create DueChecker(clock). overdue(due) compares due with clock.today(). Supply a FixedClock and test that a book due today is not overdue.

ACCEPTANCE CHECKS

[ ] Yesterday is overdue.

[ ] Today is not overdue.

[ ] Tomorrow is not overdue.

A NUDGE, IF YOU NEED ONE

Use datetime.date and inject a clock object instead of calling date.today inside the service.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DEPENDENCIES AND TEST DOUBLES

lesson_17_solution.py

```
from datetime import date

class FixedClock:
    def today(self):
        return date(2026, 10, 24)

class DueChecker:
    def __init__(self, clock):
        self.clock = clock

    def overdue(self, due):
        return due < self.clock.today()

checker = DueChecker(FixedClock())
assert checker.overdue(date(2026, 10, 23))
assert not checker.overdue(date(2026, 10, 24))
assert not checker.overdue(date(2026, 10, 25))
```

WHY IT WORKS

The fixed date makes the test reproducible next month and next year. The business rule is strict comparison: a due date becomes overdue on the following calendar date. Production systems must also define the relevant time zone.

TAKE IT FURTHER

In Python for Backend, revisit dependencies and test doubles. Add a sender that raises an error and decide whether your service should retry, report failure or record pending work.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Design principles and strategies

THE GOAL

Refactor around a reason to change.

SOLID is a set of design prompts: one responsibility, extension through stable boundaries, substitutable implementations, focused interfaces and dependencies on useful abstractions. Apply them to real change pressures. A Strategy supplies interchangeable behaviour; a function can be a strategy when an object would add no state or lifecycle.

lesson_18_example.py

```
def standard_days(member):
    return 14

def student_days(member):
    return 21

class LoanPolicy:
    def __init__(self, days_for):
        self.days_for = days_for

    def duration(self, member):
        return self.days_for(member)

assert LoanPolicy(standard_days).duration("Asha") == 14
assert LoanPolicy(student_days).duration("Asha") == 21
```

The policy depends on a callable, so a new rule does not require editing duration. This is useful when rules vary; it is unnecessary indirection if there will only ever be one trivial rule. The names explain the domain more than a pattern label would.

WATCH FOR THIS

Do not create a class for every noun or an interface for every function. Pattern names are vocabulary for tradeoffs, not a checklist of features to add.

Your turn: Design principles and strategies

THE CHALLENGE

Build a ReceiptRenderer that receives a formatting function. Supply plain and bullet formats. Keep data storage and sending outside this renderer.

ACCEPTANCE CHECKS

[ ] Plain format joins titles with a comma and space.

[ ] Bullet format puts each title on its own prefixed line.

[ ] Changing format requires no edits to ReceiptRenderer.

A NUDGE, IF YOU NEED ONE

Use a function parameter and store it; no base class is needed.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DESIGN PRINCIPLES AND STRATEGIES

lesson_18_solution.py

```
class ReceiptRenderer:
    def __init__(self, format_titles):
        self.format_titles = format_titles

    def render(self, titles):
        return self.format_titles(titles)

def plain(titles):
    return ", ".join(titles)

def bullets(titles):
    return "\n".join("- " + title for title in titles)

assert ReceiptRenderer(plain).render(["Git", "Python"]) == "Git, Python"
assert ReceiptRenderer(bullets).render(["Git"]) == "- Git"
```

WHY IT WORKS

The renderer has one clear reason to change: rendering requirements. A separate repository or sender can change for different reasons. This is a practical reading of responsibility, not a rule that every class must have only one method.

TAKE IT FURTHER

Compare Strategy with an Adapter, which translates one interface into another, and a Decorator, which wraps behaviour. See the architecture fieldbook for boundaries across modules.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Multiple inheritance and MRO

THE GOAL

Trace cooperative super calls without guessing.

With multiple inheritance, Python computes a method resolution order (MRO) that respects declared ordering and avoids visiting the same base twice. super() continues after the current class in that order; it does not always mean one fixed parent. Cooperative methods need compatible signatures and a deliberate stopping point.

lesson_19_example.py

```
class End:
    def steps(self):
        return ["end"]

class Audit(End):
    def steps(self):
        return ["audit"] + super().steps()

class Cache(End):
    def steps(self):
        return ["cache"] + super().steps()

class Service(Audit, Cache):
    pass

assert Service().steps() == ["audit", "cache", "end"]
```

The order is Service, Audit, Cache, End, object. Audit delegates to Cache for this instance, even though Audit directly names End as its base. Small mixins can share optional behaviour, but deep combinations make dependencies harder to follow.

WATCH FOR THIS

Mixins that silently require undocumented attributes or incompatible constructor arguments become difficult to combine. Prefer composition when independent collaborators express the design more clearly.

Your turn: Multiple inheritance and MRO

THE CHALLENGE

Create Validate and Record mixins with cooperative run() methods returning ordered labels. Compose Pipeline(Validate, Record). Each stage must appear once.

ACCEPTANCE CHECKS

[ ] Pipeline().run() returns validate, record, done.

[ ] The terminal base appears once.

[ ] Reversing mixin order changes the sequence.

A NUDGE, IF YOU NEED ONE

Both mixins extend the same terminal base and call super().run().

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

MULTIPLE INHERITANCE AND MRO

lesson_19_solution.py

```
class End:
    def run(self):
        return ["done"]
class Validate(End):
    def run(self):
        return ["validate"] + super().run()
class Record(End):
    def run(self):
        return ["record"] + super().run()
class Pipeline(Validate, Record):
    pass
class Reversed(Record, Validate):
    pass

assert Pipeline().run() == ["validate", "record", "done"]
assert Reversed().run() == ["record", "validate", "done"]
assert [c.__name__ for c in Pipeline.__mro__][:3] == [
    "Pipeline", "Validate", "Record"]
```

WHY IT WORKS

Each implementation contributes once and delegates once. Direct calls such as End.run(self) would skip the cooperative chain. An inconsistent inheritance graph can be rejected with TypeError when the class is created.

TAKE IT FURTHER

Write out the MRO before running the program. Replace the mixins with a list of processing functions and compare which version is easier to maintain.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Descriptors and attribute lookup

THE GOAL

Recognise reusable attribute behaviour behind properties.

A descriptor is an object stored on a class that customises attribute access through __get__, __set__ or __delete__. A data descriptor defines __set__ or __delete__ and takes priority over an instance dictionary entry. Properties are data descriptors; ordinary methods use non-data descriptors to bind self.

lesson_20_example.py

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

    def __get__(self, obj, owner=None):
        if obj is None:
            return self
        return getattr(obj, self.storage)

    def __set__(self, obj, value):
        setattr(obj, self.storage, value.upper())

class Label:
    text = Upper()
    def __init__(self, text):
        self.text = text

assert Label("python").text == "PYTHON"
```

The descriptor itself is shared, so instance-specific values are stored on obj. __set_name__ learns the public attribute name during class creation. Returning self for class access makes Label.text available for introspection.

WATCH FOR THIS

Implement a property first unless repeated attribute behaviour justifies a descriptor. __getattribute__ intercepts much broader lookup and is easy to make recursively call itself.

Your turn: Descriptors and attribute lookup

THE CHALLENGE

Write a NonNegative descriptor accepting only non-negative integers. Attach it to Inventory.quantity and confirm separate instances retain separate quantities.

ACCEPTANCE CHECKS

[ ] 0 and positive integers are accepted.

[ ] -1 and True raise ValueError.

[ ] Updating one instance does not update another.

A NUDGE, IF YOU NEED ONE

Do not store each quantity on the shared descriptor; store it on the instance.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DESCRIPTORS AND ATTRIBUTE LOOKUP

lesson_20_solution.py

```
class NonNegative:
    def __set_name__(self, owner, name):
        self.key = "_" + name
    def __get__(self, obj, owner=None):
        return self if obj is None else getattr(obj, self.key)
    def __set__(self, obj, value):
        if type(value) is not int or value < 0:
            raise ValueError("non-negative integer required")
        setattr(obj, self.key, value)

class Inventory:
    quantity = NonNegative()
    def __init__(self, quantity):
        self.quantity = quantity

a, b = Inventory(3), Inventory(7)
a.quantity = 2
assert (a.quantity, b.quantity) == (2, 7)
```

WHY IT WORKS

The descriptor centralises a validation rule that several classes could reuse. This small version expects construction to set the value before reading it and one descriptor instance per declared field.

TAKE IT FURTHER

Read the Descriptor Guide in the references. Explain lookup order: data descriptor, instance dictionary, non-data descriptor or class attribute, then __getattr__ after normal lookup fails.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Construction hooks and metaclasses

THE GOAL

Choose the smallest tool for class customisation.

Classes are themselves objects. __new__ creates an instance; __init__ prepares it. __init_subclass__ reacts when subclasses are defined. A metaclass controls class creation itself and is rarely needed in application code. Use a decorator or subclass hook before introducing a custom metaclass.

lesson_21_example.py

```
class Plugin:
    registry = {}

    def __init_subclass__(cls, *, key, **kwargs):
        super().__init_subclass__(**kwargs)
        if key in Plugin.registry:
            raise ValueError("duplicate plugin key")
        Plugin.registry[key] = cls

class TextPlugin(Plugin, key="text"):
    pass

assert Plugin.registry["text"] is TextPlugin
assert isinstance(TextPlugin, type)
```

The hook runs at class definition, before any TextPlugin instance exists. This registry is local process state, not persistent storage or automatic plugin discovery. Registration happens only when the defining module executes.

WATCH FOR THIS

Combining classes with unrelated metaclasses can cause conflicts. A clever creation mechanism is a maintenance cost unless it solves a clear recurring problem.

Your turn: Construction hooks and metaclasses

THE CHALLENGE

Create a metaclass Named that rejects class namespaces without a non-empty string label. Define one valid class and prove a missing label is rejected at class creation.

ACCEPTANCE CHECKS

[ ] A class with label="text" is created.

[ ] A class without a label raises TypeError.

[ ] Validation happens before an instance is made.

A NUDGE, IF YOU NEED ONE

Subclass type and override __new__(mcls, name, bases, namespace).

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CONSTRUCTION HOOKS AND METACLASSES

lesson_21_solution.py

```
class Named(type):
    def __new__(mcls, name, bases, namespace):
        label = namespace.get("label")
        if not isinstance(label, str) or not label.strip():
            raise TypeError("label required")
        return super().__new__(mcls, name, bases, namespace)

class TextFormat(metaclass=Named):
    label = "text"

assert TextFormat.label == "text"
try:
    class Missing(metaclass=Named):
        pass
except TypeError:
    pass
else:
    raise AssertionError("missing label accepted")
```

WHY IT WORKS

This deliberately narrow rule requires each class to declare its own label; an inherited label does not satisfy it. The example teaches the creation stage, not a recommendation to impose this pattern across a project.

TAKE IT FURTHER

Reimplement label validation with __init_subclass__. For immutable subclasses such as a normalised string, investigate __new__ and explain why __init__ cannot change the already-created string value.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Memory, copying and slots

THE GOAL

Reason about ownership before optimising objects.

Assignment creates another reference, not another object. A shallow copy duplicates an outer container; deepcopy recursively copies supported object graphs while tracking shared references. __slots__ can restrict per-instance attributes and reduce overhead for suitable classes. Measure real workloads before choosing it for performance.

lesson_22_example.py

```
from dataclasses import dataclass

@dataclass(slots=True)
class Point:
    x: int
    y: int

point = Point(2, 3)
assert not hasattr(point, "__dict__")
try:
    point.z = 4
except AttributeError:
    pass
else:
    raise AssertionError("unexpected attribute")
```

This simple slotted class has no instance dictionary and does not accept an undeclared z. Subclasses can regain a dictionary unless designed otherwise. Slots do not freeze x and y, guarantee a speedup, or make the object thread-safe.

WATCH FOR THIS

Python memory-management details vary by implementation. Do not depend on immediate destruction after del; use explicit cleanup and avoid assuming the GIL makes compound state changes atomic.

Your turn: Memory, copying and slots

THE CHALLENGE

Compare aliasing, a shallow list copy and deepcopy for a nested reading plan. Mutate the inner list and state which views should change.

ACCEPTANCE CHECKS

[ ] An alias is the original object.

[ ] A shallow copy shares the inner list.

[ ] A deep copy retains the original inner contents.

A NUDGE, IF YOU NEED ONE

Use is to inspect identity and == to inspect values.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

MEMORY, COPYING AND SLOTS

lesson_22_solution.py

```
from copy import deepcopy

plan = [["Python"]]
alias = plan
shallow = plan.copy()
deep = deepcopy(plan)
plan[0].append("Git")

assert alias is plan
assert shallow is not plan
assert shallow[0] is plan[0]
assert shallow == [["Python", "Git"]]
assert deep == [["Python"]]
assert deep[0] is not plan[0]
```

WHY IT WORKS

The outer shallow copy is new but its element refers to the same inner list. The deep copy owns a separate inner list in this example. Real objects holding sockets, locks or external resources need explicit ownership policies; blindly deep-copying them is rarely meaningful.

TAKE IT FURTHER

Profile a large set of ordinary versus slotted records with a suitable memory tool. Record interpreter and workload, and avoid claiming results you have not measured.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Capstone: a lending domain

THE GOAL

Design the small object model before adding storage.

Build a library lending system for a single process. A Book record identifies one physical copy by code. A Library owns its catalogue and active loans. Rules: codes are unique, borrowers are non-empty, only available copies may be lent, and only active loans may be returned. Keep UI and storage outside the domain.

lesson_23_example.py

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Book:
    code: str
    title: str

class Catalogue:
    def __init__(self):
        self._books = {}
    def add(self, book):
        if book.code in self._books:
            raise ValueError("duplicate code")
        self._books[book.code] = book
    def get(self, code):
        return self._books[code]

catalogue = Catalogue()
catalogue.add(Book("P1", "Python"))
assert catalogue.get("P1").title == "Python"
```

The dictionary indexes physical copies by code. Two copies may share a title but must have different codes. The complete capstone.py companion strengthens input validation and exposes behaviour through a Library class; this page isolates the catalogue decision.

WATCH FOR THIS

This register alone does not know whether a code exists in a catalogue. A complete service must compose those checks instead of assuming this isolated component enforces every rule.

Your turn: Capstone: a lending domain

THE CHALLENGE

Build a LoanRegister with lend(code, borrower), return_book(code) and borrower_for(code). Reject duplicate loans and invalid returns. Catalogue membership will be checked by the complete Library.

ACCEPTANCE CHECKS

[ ] Lending P1 to Asha records Asha.

[ ] Lending P1 again is rejected without replacement.

[ ] Returning P1 makes borrower_for return None.

A NUDGE, IF YOU NEED ONE

A dictionary maps active copy codes to borrower names. Validate before changing it.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CAPSTONE: A LENDING DOMAIN

lesson_23_solution.py

```
class LoanRegister:
    def __init__(self):
        self._loans = {}
    def lend(self, code, borrower):
        if not isinstance(borrower, str) or not borrower.strip():
            raise ValueError("borrower required")
        if code in self._loans:
            raise ValueError("already lent")
        self._loans[code] = borrower.strip()
    def return_book(self, code):
        if code not in self._loans:
            raise ValueError("not on loan")
        return self._loans.pop(code)
    def borrower_for(self, code):
        return self._loans.get(code)

loans = LoanRegister()
loans.lend("P1", "Asha")
assert loans.borrower_for("P1") == "Asha"
assert loans.return_book("P1") == "Asha"
assert loans.borrower_for("P1") is None
```

WHY IT WORKS

The active-loan mapping represents only current loans, not history. Returning a copy removes its entry. Domain objects express allowed transitions; they should not prompt for input, print menus or open database connections.

TAKE IT FURTHER

Run capstone.py and test_capstone.py from the kit. Draw the catalogue and active-loan mappings before and after each command, including one rejected command.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Capstone: boundaries and review

THE GOAL

Evaluate the design and plan a safe next increment.

Finish by separating domain behaviour from presentation and persistence. The companion library runs entirely in memory and its tests cover valid and rejected transitions. A future database adapter needs transactional constraints; a web adapter needs request validation and authentication. Those are new boundaries, not evidence that the in-memory object is production-ready.

lesson_24_example.py

```
class MemoryTitles:
    def __init__(self):
        self._items = {}
    def add(self, code, title):
        self._items[code] = title
    def get(self, code):
        return self._items[code]

class TitleService:
    def __init__(self, repository):
        self.repository = repository
    def label(self, code):
        return "Read: " + self.repository.get(code)

repo = MemoryTitles()
repo.add("P1", "Python")
assert TitleService(repo).label("P1") == "Read: Python"
```

The service depends on the repository operation it needs. The in-memory adapter deliberately has a small contract. Replacing it with SQL requires preserving error meaning and deciding who owns a transaction; matching method names alone is insufficient.

WATCH FOR THIS

Adding persistence, retries or a web server changes the failure model. Do not serialize arbitrary untrusted pickle input; define and validate an explicit data format instead.

Your turn: Capstone: boundaries and review

THE CHALLENGE

Write a summary(library) function using only the complete capstone's public API. Return available and borrowed counts. Check a library before lending, after lending and after returning.

ACCEPTANCE CHECKS

[ ] Two new copies give available=2, borrowed=0.

[ ] After one loan, counts are 1 and 1.

[ ] After return, counts are 2 and 0.

A NUDGE, IF YOU NEED ONE

Use len(library) and library.active_loans(); do not access underscore attributes.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

CAPSTONE: BOUNDARIES AND REVIEW

lesson_24_solution.py

```
# This exercise uses the companion capstone module.
from capstone import Book, Library

def summary(library):
    borrowed = len(library.active_loans())
    return {"available": len(library) - borrowed,
            "borrowed": borrowed}

library = Library()
library.add(Book("P1", "Python"))
library.add(Book("G1", "Git"))
assert summary(library) == {"available": 2, "borrowed": 0}
library.lend("P1", "Asha")
assert summary(library) == {"available": 1, "borrowed": 1}
library.return_book("P1")
assert summary(library) == {"available": 2, "borrowed": 0}
```

WHY IT WORKS

The report computes a view without changing the domain. It can survive changes to internal storage because it uses the public contract. In a concurrent service, multiple reads may require a consistent snapshot; this single-process teaching example does not provide one.

TAKE IT FURTHER

Graduate project: add due dates using an injected clock, test boundary dates, then add a SQLite repository. Read Python for Backend and Software Architecture before introducing APIs or distributed services.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sources & next steps

Use these official references when you need the full API contract or a deeper explanation. Documentation can evolve; check the version used by your own environment.

Python language: classes and objects
https://docs.python.org/3/tutorial/classes.html

Special methods and object contracts
https://docs.python.org/3/reference/datamodel.html

Data classes
https://docs.python.org/3/library/dataclasses.html

Descriptors: advanced attribute behaviour
https://docs.python.org/3/howto/descriptor.html

WHAT THE CHECKS ESTABLISH

All 48 companion scripts were executed independently during preparation. Their assertions check the small fixtures shown here. They do not establish production readiness, security under attack, browser accessibility or performance at scale.

YOUR NEXT PROJECT

Choose a small problem you can explain. Write its inputs, outputs and success checks. Build a minimal version, test one failure path, and ask another learner to follow your README. Record what you would improve next.

Keep your notes. Keep experimenting.

A route you can finish

WEEK 1 / FOUNDATIONS / LESSONS 1–6

Allow three or four sessions of 45–60 minutes. Run the Python refresher, create two objects and explain self aloud. Practise independent mutable state and a property with a rejected update. Move on when you can create a class without copying a template.

WEEK 2 / COLLABORATION / LESSONS 7–12

Compare composition with inheritance, then write two implementations of one interface. Use a Protocol, explain identity versus equality and build a data class. Milestone: justify whether a new feature needs a function, record, collaborator or subclass.

WEEK 3 / RELIABILITY / LESSONS 13–18

Implement a reusable iterable and a context manager. Test successful and rejected state transitions. Inject a clock or sender. Milestone: a deliberate defect must make your test fail, and the repaired implementation must pass.

WEEK 4 / ADVANCED PRACTICE / LESSONS 19–24

Trace an MRO, build one descriptor and compare subclass hooks with a metaclass. Complete the lending project and inspect its tests. Advanced mechanisms are recognition tools: you do not need to use all of them in your final project.

YOUR STUDY LOOP

Predict, run, explain, change, test. Keep a mistake log with the symptom, cause and corrected mental model. Suggested pacing is flexible; repeat a lesson when you cannot explain why its solution works.

The vocabulary, in plain language

OBJECT / CLASS / INSTANCE

An object has identity, type and value or state. A class defines a type and its behaviour. An instance is an object of a particular class; classes themselves are objects too.

ATTRIBUTE / METHOD / SELF

An attribute is a named value accessed on an object. A method is callable behaviour accessed through a class or instance. self names the instance parameter in an ordinary instance method.

ENCAPSULATION / INVARIANT / ABSTRACTION

Encapsulation provides a coherent interface around implementation choices. An invariant is a rule valid state must obey. An abstraction exposes the decisions a caller needs while hiding irrelevant details.

COMPOSITION / INHERITANCE / POLYMORPHISM

Composition assembles collaborators. Inheritance derives a class from bases. Polymorphism lets one caller use different implementations of the same behavioural contract.

PROTOCOL / DESCRIPTOR / MRO

A typing Protocol describes a structural interface. A descriptor customises attribute operations. The method resolution order determines the class search order used by inheritance and cooperative super calls.

VALUE OBJECT / ENTITY / DEPENDENCY

A value object is compared by meaningful values. An entity has a continuing domain identity even as details change. A dependency is a collaborator required to perform work; injection supplies it explicitly.

Read the right book next

PYTHON: THE PRACTICAL NOTEBOOK

Foundation bridge for variables, loops, functions, files and tests. Read before lesson 2 if lesson 1 feels unfamiliar.
https://depriver.tech/archive/python-the-practical-notebook.pdf

PYTHON WITH DSA

Revisit Lists and copying, Hash maps and counting, and collection costs beside lessons 4, 11, 13 and 22.
https://depriver.tech/archive/python-dsa.pdf

PYTHON FOR BACKEND

Continue from lessons 10, 17 and 24 into service boundaries, dependencies and test doubles.
https://depriver.tech/archive/python-backend.pdf

SOFTWARE ARCHITECTURE: THE PRACTICAL FIELDBOOK

Extend composition and dependency decisions into modular boundaries, ports and adapters, and data ownership.
https://depriver.tech/archive/software-architecture-fieldbook.pdf

GIT AND AUTOMATION

Use Git for small, recoverable study commits and Automation for file-oriented projects.
https://depriver.tech/archive/git-the-practical-notebook.pdf
https://depriver.tech/archive/python-automation.pdf

Prove what you understand

BEGINNER CHECKPOINT

Without looking back, create a class, instantiate it twice, change one instance and explain why the other is unchanged. Explain self, __init__, class attributes and a mutable default argument.

INTERMEDIATE CHECKPOINT

Write two interchangeable collaborators and tests for a rejected operation. Explain when composition is clearer than inheritance. Distinguish == from is and show why a frozen object containing a list is not deeply immutable.

ADVANCED CHECKPOINT

Trace Service.__mro__ from lesson 19, explain a descriptor lookup and identify when a metaclass is unnecessary. Describe the limits of slots, annotations, fake adapters and in-memory state under concurrency.

CAPSTONE REVIEW / 20 POINTS

Award 0-4 each for domain rules, clear public API, failure-path tests, separation of effects, and a reproducible README. Aim for at least 16 with no missing category. This is a self-review rubric, not a certification or guarantee of production readiness.

NEXT INCREMENT

Add due dates with a fixed-clock test, then consider SQLite persistence. Preserve the current public contract or document its change. Define how concurrent lending is rejected atomically before exposing a network API.

PRIMARY REFERENCES FOR FURTHER STUDY

Typing protocols: https://docs.python.org/3/library/typing.html#typing.Protocol
Abstract bases: https://docs.python.org/3/library/abc.html
Testing: https://docs.python.org/3/library/unittest.html