

Topic 04

University Computer Science

What this course is, and how to use it

This is the fourth course in University Computer Science: **object-oriented programming** – how to grow a small script into a system that other people can safely change: objects that own their state, interfaces that hide their machinery, and tests that hold the line while you refactor.

Scope limit, stated once: a study course, not a degree; Python 3 is the vehicle, and every program fragment is original, self-contained, and was executed to produce the outputs shown.

- Units 4.1–4.2 build the object (classes, references, walls); 4.3–4.4 connect them; 4.5–4.6 keep the system alive.

The design habit: **every class is a contract with its callers** – each unit names the contract, and every example is a program you can run.

Objects and classes

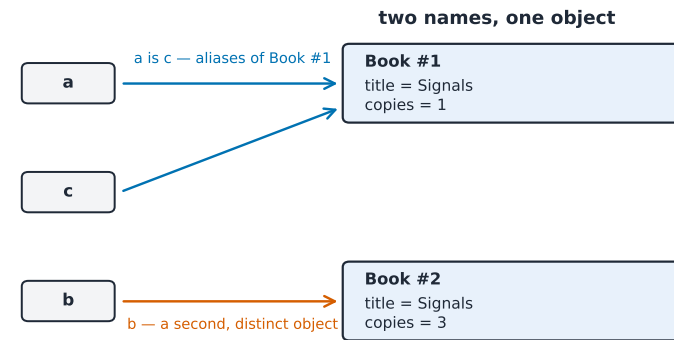
Why it matters. Programs fail when state lives everywhere: a **class** bundles state and behaviour so there is exactly one place to look. From here on, every system you build is classes talking.

The idea, with a picture. A class is a blueprint; an **instance** (an **object**) is one thing built from it. `__init__` runs at construction and seeds the **instance attributes**; methods take **self**, the instance itself. **Class attributes** live on the class and are shared by all instances. Assignment never copies: a variable is a **reference** to an object, so two names can be **aliases** of one object.

```
class Book:
    next_id = 1                # class attribute, shared

    def __init__(self, title, copies):
        self.id = Book.next_id; Book.next_id += 1
        self.title = title
        self.copies = copies   # instance attribute

    def take(self):
        self.copies -= 1; return self.copies
```



Two names on one object: a and c alias the same Book, b is a second, equal-but-distinct object

The lookup order (a derivation). `obj.x` resolves by walking outward: the instance dict first (`obj.__dict__`), then the class dict, then the base classes. That single rule explains the classic trap: assigning `obj.next_id = 9` does not touch the class – it shadows it for that one instance. `id` above lands in the instance dict; `next_id` stays in the class dict, so the books get ids 1, 2, 3, ... and `Book.next_id` always holds the next free number.

Worked example 1 – one class, traced. With `Book` above:

```
a = Book('Signals', 3)
b = Book('Signals', 3)
c = a
a.take(); a.take()
```

State after: `a.copies` is 1, `b.copies` is 3 (untouched), and because `c is a`, `c.copies` is also 1 – one object, two names. `a == b` is **False** (default equality is identity) while `a is c` is **True**: **equality** compares values (once you define how), **identity** (`is`) compares objects.

Worked example 2 – the shared counter. After `a` and `b` above, `Book.next_id`, `a.next_id` and `b.next_id` all read 3 – the counter belongs to the class, so every caller agrees. Write `a.next_id = 9` and only `a` reads 9: the class still says 3, and the next book still gets id 3.

Common mistakes. Forgetting `self` in a method header. Mutating a class attribute through an instance and wondering why others disagree. Expecting `d = c` to copy – it aliases.

Summary. Class = blueprint, instance = thing; instance dict shadows class dict; assignment aliases, never copies; `is` is identity, `==` is (defined) equality.

Encapsulation

Why it matters. Every class owns an **invariant** – a statement that is true between calls (a copy is either on the shelf or out to one borrower, nothing else).

Encapsulation is the discipline of letting that statement stay true: state is private, changes pass through methods that check it.

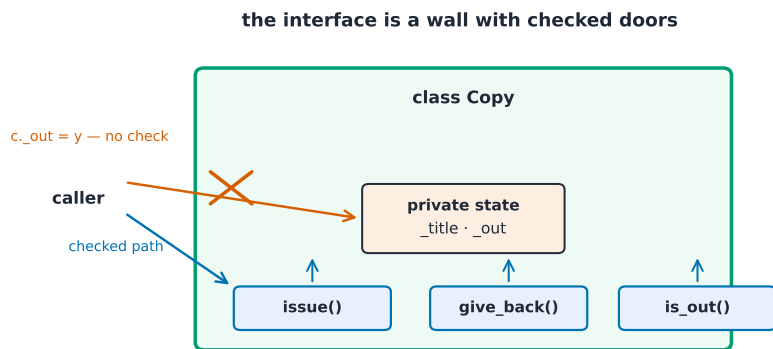
The idea, with a picture. By convention a leading underscore marks internal state (`_copies`); callers use the public methods – the **interface** boundary. Validate at the gate: the constructor and every mutating method re-establish the invariant or refuse.

```
class Copy:
    def __init__(self, title):
        if title == '': raise ValueError('empty title')
        self._title = title; self._out = False

    def issue(self):
        if self._out: raise ValueError('already out')
        self._out = True

    def give_back(self):
        if not self._out: raise ValueError('not out')
        self._out = False

    def is_out(self):          # public reader, no wall breach
        return self._out
```



Public methods form a wall: every state change passes the checks; the fields behind the wall are private

The derivation (why one gate is enough). If every path that touches `_out` goes through `issue/give_back`, the invariant (`_out` is exactly `True` or `False`) is provable from two small methods. Add one unchecked writer and the proof is gone – the number

of places to check grows with every caller, and so does the bug rate.

Worked example 1 – the wall holds. `c.issue()` succeeds; `c.issue()` again raises `ValueError` – the caller cannot reach an impossible state, and after the failed call the object is unchanged (check-first, so no rollback is needed). `c.give_back()`; `c._out` is `False`, and `c.issue()` works again: exactly the states `True / False`, nothing else, forever.

Worked example 2 – the cost of a leak. A teammate in a hurry writes `c._out = 'y'`. Nothing stops them – underscore is convention – and now `is_out()` answers 'y': truthy but not `True`. One caller (if `c.is_out()`;) shelves the copy as out; another (if `c.is_out()` is `True`;) shelves it as available – two readers disagree about one object, a state no method could ever produce. Repair cost scales with every reader of `_out`; the validation cost was two `if` statements. That asymmetry is the whole argument.

Common mistakes. Returning the internal list (`return self._loans`) – callers mutate your private state; return a copy. Validating in the constructor but not in the setter. Writing `get/set` pairs for everything: expose behaviour (`available()`), not storage.

Summary. Underscore marks intent; methods validate at the gate; one checked path per change keeps invariants provable; expose behaviour, not fields.

Composition and inheritance

Why it matters. Objects connect two ways: **inheritance** (is-a: a `Member` is a `Person`) and **composition** (has-a: a `Library` has a `Catalog`). Choosing wrong couples classes so tightly that a base-class fix breaks its children – the **fragile base class** problem.

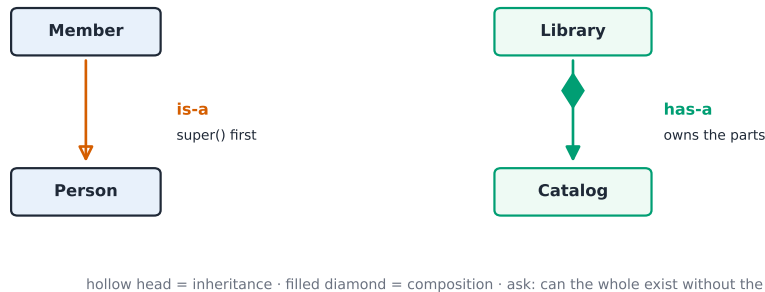
The idea, with a picture. Inheritance chains initialisation: `super().__init__(..)` walks up the chain, base first. Composition delegates: the whole object owns parts and forwards calls.

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

class Member(Person):
    def __init__(self, name, card_no):
        super().__init__(name)          # base first
        self._card = card_no; self._loans = 0

    def borrow(self):
        self._loans += 1; return self._loans
```

is-a inherits upward, has-a owns inward



*Two connection kinds: inheritance says is-a (hollow triangle to the base);
composition says has-a (filled diamond to the part)*

The derivation (initialisation order). `Member('Ana', 7)` runs `Member.__init__`, whose first act is `super().__init__('Ana')` – so `Person.__init__` completes before any `Member` line executes. Order is always base-first down the chain; skip the `super()` call and `self._name` does not exist when `Member` needs it, failing with a clear `AttributeError` instead of corrupting quietly.

Worked example 1 – composition delegates. A `Library` has a `Catalog` (a list of `Copy`):

```
class Library:
    def __init__(self):
        self._shelves = []          # composition

    def add(self, copy):
        self._shelves.append(copy)

    def available(self):
        return sum(1 for c in self._shelves if not c.is_out())
```

After adding 5 copies and issuing 2, `available()` is 3 – the library owns no borrowing logic; it asks its part. Swap the list for a database listlike and `Library` is untouched: parts change, the whole keeps its interface.

Worked example 2 – the fragile base. Base `Receipt.total()` sums items; a subclass overrides by copying the summing code and subtracting 10. The base then gains a loyalty line – every discount receipt silently loses it: the copy is a fork. Repair: `return super().total() - 10` – override by delegating. Inheritance for is-a, composition for everything else.

Common mistakes. Inheriting to reuse two methods (composition or a module function instead). Forgetting `super().__init__`. Duplicating base logic in an override –

the fork that goes stale.

Summary. is-a inherits (and calls `super()`); has-a composes (and delegates); never copy base logic into an override.

Interfaces and polymorphism

Why it matters. Big systems survive change when callers depend on what a thing does, not which class it is. An **interface** names the behaviour; **polymorphism** is one call site running many implementations.

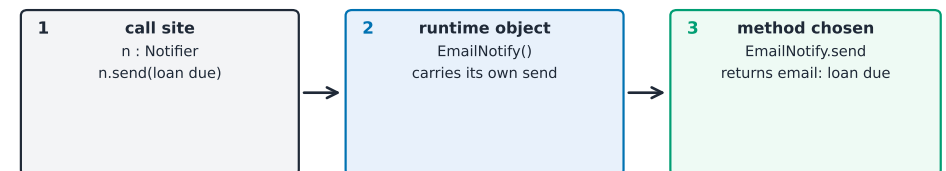
The idea, with a picture. Python is dynamically typed: any object with the right methods fits – **duck typing**. To make the contract explicit, write a Protocol (or `abc.ABC`) naming the methods.

```
class Notifier(Protocol):
    def send(self, text: str) -> str: ...

class EmailNotify:
    def send(self, text): return f'email: {text}'

class SmsNotify:
    def send(self, text): return f'sms: {text}'
```

one call, three steps, one winner



the annotation names the interface — the object decides the method; swap in `SmsNotify` and only step 2 and 3 change

One call, three steps: the variable names the interface, the object carries the runtime type, dispatch picks that type's send

The derivation (dynamic dispatch). When `n.send('due')` runs, the attribute lookup of 4.1 resolves on the **runtime type** – the object itself, not the variable's annotation. So one loop body dispatches to a different `send` per element. That is **dynamic dispatch**: the call site is written once, the selection happens per object at call time.

Worked example 1 – one loop, three senders. With a third sender `QueueNotify` of the same shape and `team = [EmailNotify(), SmsNotify(), QueueNotify()]`:

```
for n in team:
    print(n.send('loan due'))
```

Three lines out – `email: loan due, sms: loan due, queue: loan due` – and adding a fourth sender touches no existing line: the caller was never told the concrete classes. This is the **open/closed** shape: open to new types, closed to edit.

Worked example 2 – substitution and its trap. A substitute that honours the contract slots in invisibly. One that **narrows** it fails silently: a `SilentNotify.send` that raises on an empty string breaks a caller whose contract said *send never raises*. The rule of **substitution** (Liskov): preconditions may only widen, guarantees only grow.

Common mistakes. `isinstance` chains – replace with a method each class implements. An override that adds a precondition or drops a guarantee: the caller cannot tell, until it breaks.

Summary. Name behaviour with a Protocol; depend on the name; dispatch follows the runtime type; substitutes must honour the whole contract.

Exceptions and error handling

Why it matters. Errors travel. A function three frames down cannot fix a failure, but it must not ignore it. **Exceptions** carry an error to the frame that can decide, running each frame’s cleanup on the way.

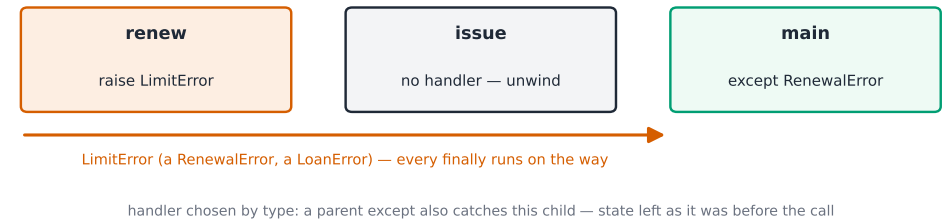
The idea, with a picture. `raise` throws; `try/except` catches by type; `finally` runs on **every** path. A **custom hierarchy** lets callers catch precisely: broad base, specific children.

```
class LoanError(Exception): ...
class RenewalError(LoanError): ...
class LimitError(RenewalError): ...

class Loan:
    def __init__(self, days):
        self.days = days

def renew(loan, extra):
    if loan.days + extra > 30:
        raise LimitError(f'{loan.days}+{extra} exceeds 30')
    loan.days += extra
    return loan.days
```

the raise travels down the stack until caught



Raise in renew unwinds issue and main until a matching handler; finally runs on the way out, on every path

The derivation (status values vs exceptions). A return of `-1` is a normal value: the compiler and runtime attach nothing to it, so a caller that forgets to check keeps computing with nonsense – the failure spreads silently. An exception, once raised, cannot be ignored: it unwinds frame by frame (running each **finally**) until some **except** matches; if none does, the program stops loudly. Ignorable errors get ignored; unignorable errors get handled. Choose **status values** for expected, caller-adjacent outcomes (a miss in a lookup), **exceptions** for broken contracts and failures the immediate caller cannot fix.

Worked example 1 – precise catch. `renew(Loan(20), 15)` raises `LimitError` with message `20+15 exceeds 30`; the caller catches `RenewalError` (a parent) and the loan still reads 20: check-first means failed calls leave state untouched. Hierarchy is a filter – the caller picks the grain.

Worked example 2 – cleanup on both paths. A lock guard:

```
locked = True
loan = Loan(28)
try:
    total = renew(loan, 10)           # raises
except RenewalError:
    kept = 'handled'
finally:
    locked = False
```

Trace: `renew` raises; `except` runs (`kept = 'handled'`); `finally` runs; afterwards `locked` is `False` and the loan still reads 28 – released on the failing path. On the success path `finally` runs just the same. Cleanup that must happen goes in **finally** (or a `with` block), never after the last statement.

Common mistakes. `except:` bare – catches `KeyboardInterrupt` and hides real bugs. `except ... pass` – the failure vanishes and state is half-updated. Catching `Exception` when you meant one specific child.

Summary. Raise on broken contracts, catch at the frame that can decide, **finally** for cleanup; a small hierarchy buys precise handling; failed calls must leave state unchanged.

Testing and design quality

Why it matters. Refactoring – changing structure without changing behaviour – is only safe when behaviour is pinned. **Unit tests** are that pin, and the good ones test behaviour, not methods.

The idea, with a picture. One test = one behaviour: **arrange** the objects, **act** one call, **assert** the observable outcome. Behaviours, not methods: **renew** past the limit raises is a behaviour; **test_days** is a method.

```
def test_renew_ok():
    loan = Loan(20)                # arrange
    out = renew(loan, 7)           # act
    assert out == 27               # assert

def test_renew_boundary():
    assert renew(Loan(20), 10) == 30    # exactly 30 is allowed

def test_renew_limit():
    loan = Loan(20)
    try:
        renew(loan, 11); assert False
    except LimitError:
        assert loan.days == 20        # unchanged
```

each behaviour needs a pin			
behaviour ↓ · test →	test_renew_ok	test_renew_limit	test_renew_owner
renew within limit	pinned		
renew at the limit		pinned	
renew, wrong owner			pinned
return a copy twice			

Behaviours down, tests across: each cell is a behaviour a test pins; the empty cell is next to write

The derivation (why behaviour-first). A method-named test breaks on every rename and survives every behaviour change – noise without safety; a behaviour-named test does the opposite. Coverage follows the behaviour list: every branch of the contract is one row, and the grid shows the next hole.

Worked example 1 – three behaviours, three tests. For **renew**: within the limit ($20 + 7 = 27$), exactly at the limit ($20 + 10 = 30$ is allowed, because the check is > 30), and past it ($20 + 11$ raises, state unchanged). The boundary test catches the classic slip of writing $\geq$ for $>$. Running them prints **3 passed**.

Worked example 2 – refactor under green. The loan’s fine calculation moves out of `Loan` into a `FineCalculator` class (**single responsibility principle** : one reason to change). The three tests are untouched, still **3 passed** – behaviour preserved while structure moved: **coupling** went down, **cohesion** went up.

Common mistakes. Asserting private internals – refactors break such tests. Shared mutable test state (test B passes only after test A ran). Chasing line coverage on getters while the limit branch has no test.

Summary. Arrange-act-assert per behaviour; tests pin behaviour so refactors are safe; SRP lowers the cost of change; low coupling and high cohesion are the direction of every move.