Module 8 of 20

Classes and objects

The class statement with __init__ and self, instance versus class attributes, the dunder methods that make a class behave like a built-in, @property and __slots__, class methods as alternative constructors, dataclasses and their options, and the design discipline of invariants and small interfaces.

A Python class is transparent in a way most languages' are not: an instance is a dictionary of attributes with a pointer to its class, a method is a function whose first parameter receives the instance, and every operator is a call to a double-underscore method the class may define. This module builds classes on that understanding — no magic, just a protocol — from `__init__` and `self` through the data model, properties, class methods and dataclasses, to the judgement of when a class is worth writing at all: when it protects an invariant behind a small interface.

Defining classes covers the statement, `self`, instance attributes, the shared-mutable class-attribute trap and `__repr__`. Dunder methods gives representation, equality and hashing, ordering with `total_ordering`, the collection dunders, arithmetic with reflected operators and `NotImplemented`. Properties and encapsulation covers the underscore conventions and name mangling, `@property` with setters, `cached_property`, read-only attributes and `__slots__`. Class and static methods covers `cls`, alternative constructors, class-level state and when a module function is better. Dataclasses gives the decorator, `field`, the options (`order`, `frozen`, `slots`, `kw_only`), `__post_init__` and the helpers. Designing a class works one example through invariants, representation, interface and tests.

The exercises are whole programs: a tally class with a class-level count, an account with a proper `__repr__`, a `Money` type with arithmetic and ordering dunders, a playlist that behaves as a collection, a temperature with validated and computed properties, a slotted point, a duration built by alternative constructors, a username validated by a static method, dataclass records sorted and replaced, a frozen point set, the inventory from the lesson and a bounded stack. The checkpoint adds a normalised fraction value type, a ledger of accounts with transfers, and a frozen product catalogue with derived fields.

CONTENTS

1 Defining classes — __init__, self, attributes and methods 14 min

2	Dunder methods — making a class behave like a built-in	15 min
3	Properties and encapsulation — the underscore, @property and __slots__	13 min
4	Class methods and static methods	12 min
5	Dataclasses — classes that are mostly data	14 min
6	Designing a class — invariants, interfaces and a worked example	14 min
✓	Checkpoint — Classes and objects	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Defining classes — __init__, self, attributes and methods

A class bundles data and the functions that operate on it into one kind of object, and Python's version is unusually transparent: an instance is a dictionary of attributes with a pointer to its class, a method is an ordinary function whose first parameter receives the instance, and `self` is nothing more than the name that parameter is given by convention. Seeing that plainly removes most of the mystery. This lesson covers the `class` statement, `__init__`, instance versus class attributes and the trap between them, methods and how `obj.method()` becomes `Class.method(obj)`, and `__repr__` as the first dunder every class should have.

The statement

```
class Account:
    """A bank account with a balance that cannot go negative."""

    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("insufficient funds")
        self.balance -= amount

acct = Account("ada", 100)
acct.deposit(50)
acct.withdraw(30)
print(acct.balance)     # 120
```

`class Account:` creates a class object bound to the name `Account`. Calling it — `Account("ada", 100)` — creates a new, empty instance and then calls `__init__` with the instance as `self` and the arguments after it. `__init__` is not a constructor in the C++ sense

(the object already exists; `__new__` made it — Module 16) but an *initialiser*: its job is to give the instance its attributes. It returns nothing.

Naming: classes are `PascalCase`, methods and attributes `snake_case`, and there is one blank line between methods, two between top-level definitions.

---

self

`self` is the instance, passed automatically. `acct.deposit(50)` is exactly `Account.deposit(acct, 50)`: attribute lookup finds the function on the class, wraps it with the instance as a *bound method*, and the call fills in `self`. That is why every method's first parameter is `self`, why forgetting it produces `TypeError: deposit() takes 1 positional argument but 2 were given`, and why `self` could be named anything — it is a convention so strong that violating it reads as an error.

Inside a method, every attribute access goes through `self`: `self.balance`, `self.deposit(…)`. A bare `balance` would be a local variable of the method, unrelated to the instance — the most common beginner bug in a class.

Instance attributes

Attributes are created by assignment, usually in `__init__`, and live in the instance's own dictionary:

```
acct.__dict__           # {'owner': 'ada', 'balance': 120}
acct.nickname = "main" # a new attribute on this instance only — legal, and us
vars(acct)              # the same dict
```

Reading a missing attribute is `AttributeError`. Because attributes are per instance, two accounts have separate balances. The discipline is to create *every* attribute in `__init__`, even if only as `None` or an empty list, so that the instance's shape is visible in one place and no method can meet an attribute that has not been set yet.

Class attributes — and the trap

An assignment in the class body (outside any method) creates a *class attribute*, shared by every instance:

```
class Account:
    interest_rate = 0.02          # class attribute: one value for all accoun
    count = 0

    def __init__(self, owner):
        self.owner = owner
        Account.count += 1        # update the shared counter through the cla
```

Reading `acct.interest_rate` finds nothing in the instance dict and falls through to the class. That is right for constants and defaults. It is wrong for a mutable default:

```
class Team:
    members = []                  # ONE list shared by every Team

    def add(self, name):
        self.members.append(name) # every team sees every name

class Team:
    def __init__(self):
        self.members = []         # a fresh list per instance
```

The same lesson as the mutable default argument (Module 4): a mutable object created once in a class body is shared. Per-instance state belongs in `__init__`.

Assigning `acct.interest_rate = 0.05` creates an *instance* attribute that shadows the class one for that object only — the class value is unchanged. `Account.interest_rate = 0.05` changes it for every instance that has not shadowed it.

Methods versus functions

A method is a function in the class body. Three consequences: it can be called on the class with an explicit instance (`Account.deposit(acct, 5)`); it can be stored and passed like any function (`f = acct.deposit; f(5)` — a bound method remembers its instance); and a function that does not use `self` at all is either a `@staticmethod` (Module 8 lesson 4) or does not belong in the class. Methods call other methods through `self`, and `__init__` may call them to set things up.

repr first

Every class should define `__repr__` before anything else, because the default `<__main__.Account object at 0x7f…>` tells you nothing in a traceback, a list, or a debug print:

```
    def __repr__(self):
        return f"Account(owner={self.owner!r}, balance={self.balance})"
```

The convention is text that looks like the constructor call. `print(acct)` uses `__str__` if defined and falls back to `__repr__`; `print([acct])` always uses `__repr__` for the elements. The next lesson covers the rest of the dunders.

Pitfalls

- Forgetting `self` in a method's parameter list, or writing `balance` for `self.balance`.
- A mutable class attribute used as per-instance state.
- Creating attributes outside `__init__` so the object's shape depends on which methods have run.
- `Account.deposit(50)` without an instance.
- Treating `__init__` as returning the object (`return self` in it is an error).
- Leaving the default `__repr__`.

Key takeaways

- `class` creates a class object; calling it makes an instance and runs `__init__(self, …)` to set attributes.
- `obj.method(args)` is `Class.method(obj, args)`; `self` is the instance and every attribute access goes through it.
- Instance attributes live in `obj.__dict__`, created by assignment — all of them in `__init__`.
- Class-body assignments are shared class attributes: fine for constants, a bug for mutable state.

Define `__repr__` to look like the constructor call; it is what tracebacks and containers show.

EXERCISE 1

Tallies with a class-level count

Write `Tally`, a named counter: `Tally(name)` starts at 0, `inc(by=1)` adds, and `__repr__` returns `Tally(name='x', value=3)`. The class keeps a **class attribute** `created` counting how many instances have been made. Run commands: `new name`, `inc name [by]`, `show name` (prints the repr), `created` (prints the class count).

Input: commands. **Output:** one line per `show` and `created`.

```
new hits
inc hits
inc hits 4
show hits
new misses
created
```

prints

```
Tally(name='hits', value=5)
2
```

INPUT 1

```
new hits
inc hits
inc hits 4
show hits
new misses
created
```

OUTPUT 1

```
Tally(name='hits', value=5)
2
```

REFERENCE SOLUTION

```python
import sys


class Tally:
    created = 0

    def __init__(self, name):
        self.name = name
        self.value = 0
        Tally.created += 1

    def inc(self, by=1):
        self.value += by

    def __repr__(self):
        return f"Tally(name={self.name!r}, value={self.value})"


tallies = {}
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "new":
        tallies[args[0]] = Tally(args[0])
    elif cmd == "inc":
        tallies[args[0]].inc(int(args[1]) if len(args) > 1 else 1)
    elif cmd == "show":
        print(tallies[args[0]])
    elif cmd == "created":
        print(Tally.created)
```

EXERCISE 2

Account

Write `Account(owner, balance=0)` with `deposit(amount)`, `withdraw(amount)` that raises `ValueError` when the balance would go negative, and a `__repr__` of the form `Account(owner='ada', balance=120)`. Process commands `open owner [balance]`, `deposit owner amount`, `withdraw owner amount` (print `insufficient funds` when the exception is raised) and `show owner`.

Input: commands. **Output:** one line per `show` and per failed withdrawal.

```
open ada 100
deposit ada 50
withdraw ada 30
withdraw ada 500
show ada
```

prints

```
insufficient funds
Account(owner='ada', balance=120)
```

INPUT 1

```
open ada 100
deposit ada 50
withdraw ada 30
withdraw ada 500
show ada
```

OUTPUT 1

```
insufficient funds
Account(owner='ada', balance=120)
```

REFERENCE SOLUTION

```
import sys


class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("insufficient funds")
        self.balance -= amount

    def __repr__(self):
        return f"Account(owner={self.owner!r}, balance={self.balance})"


accounts = {}
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "open":
        accounts[args[0]] = Account(args[0], int(args[1]) if len(args) > 1 else 0)
    elif cmd == "deposit":
        accounts[args[0]].deposit(int(args[1]))
    elif cmd == "withdraw":
        try:
            accounts[args[0]].withdraw(int(args[1]))
        except ValueError:
            print("insufficient funds")
    elif cmd == "show":
        print(accounts[args[0]])
```

QUIZ

1. What does `acct.deposit(50)` translate to?

A. `deposit(acct, 50)`

B. `Account.deposit(acct, 50)`

C. `Account.deposit(50)`

D. `acct.__dict__["deposit"](50)`

2. What does this print?

```
class Team:
    members = []

a, b = Team(), Team()
a.members.append("ada")
print(b.members)
```

A. `[]`

B. `['ada']`

C. `AttributeError`

D. `None`

3. Why does `TypeError: describe() takes 0 positional arguments but 1 was given` happen?

A. The method was called with an extra argument

B. The method was defined without `self`, but the call passes the instance automatically

C. The class has no `__init__`

D. `describe` is a class attribute

4. Inside a method, what is `balance` (without `self.`)?

A. The instance attribute

B. A local variable of the method, unrelated to the instance

C. The class attribute

D. A syntax error

5. What is the recommended shape of `__repr__`?

A. A friendly sentence for users

B. Text that looks like the constructor call, e.g. `Account(owner='ada', balance=5)`

C. The memory address

D. The class name only

ANSWER KEY · DEFINING CLASSES — __INIT__, SELF, ATTRIBUTES AND METHODS

1. B Attribute lookup finds the function on the class and binds the instance as its first argument, `self`.

2. B `members` is a class attribute — one list shared by every instance. Per-instance state must be created in `__init__`.

3. B Every instance method receives the instance as its first argument; a definition without `self` cannot accept it.

4. B Attributes are reached through `self`; a bare name is looked up in the method's local scope, then enclosing and global scopes.

5. B The repr is for developers: unambiguous, showing the state, ideally re-creatable. Containers and tracebacks display it.

LESSON 2 · 15 MIN

Dunder methods — making a class behave like a built-in

Every operator and built-in function in Python is a call to a *dunder* (double-underscore) method: `a + b` is `a.__add__(b)`, `len(x)` is `x.__len__()`, `x in c` is `c.__contains__(x)`, `print(x)` is `x.__str__()`. Define the method and your class gets the operator; this is the *data model*, and it is why a user-defined type can be as pleasant to use as a list or an int. This lesson covers the dunders that matter first — representation, equality and ordering, size and containment, iteration, arithmetic, truth and calling — with the conventions (`NotImplemented`, `total_ordering`, reflected operators) that make them behave correctly.

Representation

```
class Money:
    def __init__(self, cents, currency="GBP"):
        self.cents, self.currency = cents, currency

    def __repr__(self):                     # unambiguous, for developers
        return f"Money({self.cents}, {self.currency!r})"

    def __str__(self):                      # readable, for users
        return f"{self.cents // 100}.{self.cents % 100:02d} {self.currency}"

m = Money(1250)
repr(m)        # "Money(1250, 'GBP')"
str(m)         # '12.50 GBP'
print(m)       # 12.50 GBP       — print uses __str__
print([m])     # [Money(1250, 'GBP')] — containers use __repr__
f"{m}"         # str;  f"{m!r}" for repr
```

If only `__repr__` is defined, `str` falls back to it, so `__repr__` alone is enough for most classes; add `__str__` when the user-facing text should differ. `__format__(self, spec)` lets `f"{m:>12}"` work with a custom spec.

Equality and hashing

```
    def __eq__(self, other):
        if not isinstance(other, Money):
            return NotImplemented
        return (self.cents, self.currency) == (other.cents, other.currency)

    def __hash__(self):
        return hash((self.cents, self.currency))
```

Without `__eq__`, `==` is identity. Returning `NotImplemented` (not `False`) for a foreign type lets Python try `other.__eq__(self)` and then fall back to identity, so `Money(1) == 5` is `False` without an error. `__ne__` is derived automatically. Defining `__eq__` makes the class unhashable unless `__hash__` is also defined (Module 7 lesson 4) — define it over the same fields, or leave the class unhashable deliberately if it is mutable.

Ordering

```
from functools import total_ordering

@total_ordering
class Money:
    ...
    def __lt__(self, other):
        if not isinstance(other, Money) or other.currency != self.currency:
            return NotImplemented
        return self.cents < other.cents
```

`__lt__` alone makes `sorted`, `min` and `max` work (they only use `<`). `@total_ordering` derives `__le__`, `__gt__`, `__ge__` from `__lt__` plus `__eq__`, so you write two and get six. The comparison dunders should return `NotImplemented` for types they cannot order; Python then raises `TypeError` with a clear message.

Size, containment, indexing, iteration

```
class Playlist:
    def __init__(self, tracks):
        self._tracks = list(tracks)

    def __len__(self):                  # len(p)
        return len(self._tracks)

    def __contains__(self, track):      # track in p
        return track in self._tracks

    def __getitem__(self, i):           # p[i], p[1:3]; iteration falls back
        return self._tracks[i]

    def __iter__(self):                 # for t in p — preferred over the __ge
        return iter(self._tracks)

    def __bool__(self):                 # if p:   — without it, __len__ decide
        return bool(self._tracks)
```

`__len__` makes `len()` work and, absent `__bool__`, makes an empty object falsy. `__getitem__` receiving a `slice` object is what makes `p[1:3]` work — delegating to the inner list handles it. `__iter__` should return an iterator (`iter(self._tracks)`, or a generator with `yield`, Module 11); `__reversed__`, `__setitem__` and `__delitem__` complete the mutable-sequence set.

Arithmetic

```
    def __add__(self, other):                       # self + other
        if not isinstance(other, Money) or other.currency != self.currency:
            return NotImplemented
        return Money(self.cents + other.cents, self.currency)

    def __mul__(self, factor):                      # self * 3
        if not isinstance(factor, int):
            return NotImplemented
        return Money(self.cents * factor, self.currency)

    def __rmul__(self, factor):                     # 3 * self — the reflecte
        return self * factor

    def __neg__(self):                              # -self
        return Money(-self.cents, self.currency)
```

For `a + b` Python calls `a.__add__(b)`; if that returns `NotImplemented` it tries `b.__radd__(a)`. That is how `3 * money` works: `int.__mul__` does not know `Money`, so `Money.__rmul__` is tried. Arithmetic dunders should return a *new* object and leave the operands alone; `__iadd__` (for `+=`) may mutate in place and must return `self`. `sum(monies, Money(0))` works once `__add__` exists — the start value matters because `sum` begins from `0`.

Calling and the rest

`__call__(self, *args)` makes an instance callable (`counter()`), which is how a class becomes a function with state. `__enter__` / `__exit__` make it a context manager (Module 10). `__getattr__` intercepts missing attributes (Module 16). A summary of the ones to know:

Expression	Method	Expression	Method
`repr(x)`, `str(x)`	`__repr__`, `__str__`	`x + y`, `x * y`	`__add__`, `__mul__` (+ `__r…__`)
`x == y`, `x < y`	`__eq__`, `__lt__`	`-x`, `abs(x)`	`__neg__`, `__abs__`
`hash(x)`	`__hash__`	`x += y`	`__iadd__`
`len(x)`, `bool(x)`	`__len__`, `__bool__`	`x()`	`__call__`
`x[i]`, `x[i] = v`	`__getitem__`, `__setitem__`	`with x:`	`__enter__`, `__exit__`
`for e in x`	`__iter__`	`int(x)`, `float(x)`	`__int__`, `__float__`
`e in x`	`__contains__`	`x.missing`	`__getattr__`

Pitfalls

Returning `False` instead of `NotImplemented` for a foreign type in `__eq__` / `__lt__` / `__add__`.

`__eq__` without `__hash__` on a class meant to be a key.

Mutating `self` in `__add__`; that is `__iadd__`'s job.

`__str__` without `__repr__` — containers still show the ugly default.

Forgetting `__rmul__`, so `3 * m` fails while `m * 3` works.

A `__len__` that can be zero on an object that should be truthy — add `__bool__`.

Key takeaways

Operators and built-ins dispatch to dunders; define them and the class gains the syntax.

`__repr__` looks like the constructor; `__str__` is the readable form; containers use `repr`.

`__eq__` + `__hash__` over the same fields; `__lt__` + `@total_ordering` for the rest; return `NotImplemented` for foreign types.

`__len__`, `__contains__`, `__getitem__`, `__iter__`, `__bool__` make a collection type; `__getitem__` alone gives iteration.

Binary dunders return new objects; `__r…__` handles the reflected call; `__iadd__` mutates and returns `self`.

EXERCISE 1

Money with operators

Implement `Money` holding integer cents with `__add__`, `__mul__` and `__rmul__` (by an int), `__eq__`, `__lt__` (decorated with `functools.total_ordering`), `__hash__` and `__str__` printing `12.50`. Return `NotImplemented` for foreign types. Read `n` amounts as text and print: their total via `sum(amounts, Money(0))`, the largest, the amounts sorted, `3 * first`, and the number of distinct amounts via a set.

Input: `n`, then `n` amounts with two decimals. **Output:** `total <x>`, `max <x>`, `sorted <x ...>`, `triple <x>`, `distinct <k>`.

```
3
12.50
0.75
12.50
```

prints

```
total 25.75
max 12.50
sorted 0.75 12.50 12.50
triple 37.50
distinct 2
```

INPUT 1

```
3
12.50
0.75
12.50
```

OUTPUT 1

```
total 25.75
max 12.50
sorted 0.75 12.50 12.50
triple 37.50
distinct 2
```

REFERENCE SOLUTION

```python
from functools import total_ordering


@total_ordering
class Money:
    def __init__(self, cents):
        self.cents = cents

    @classmethod
    def parse(cls, text):
        pounds, pence = text.split(".")
        return cls(int(pounds) * 100 + int(pence))

    def __add__(self, other):
        if not isinstance(other, Money):
            return NotImplemented
        return Money(self.cents + other.cents)

    def __mul__(self, factor):
        if not isinstance(factor, int):
            return NotImplemented
        return Money(self.cents * factor)

    def __rmul__(self, factor):
        return self * factor

    def __eq__(self, other):
        if not isinstance(other, Money):
            return NotImplemented
        return self.cents == other.cents

    def __lt__(self, other):
        if not isinstance(other, Money):
            return NotImplemented
        return self.cents < other.cents

    def __hash__(self):
        return hash(self.cents)

    def __str__(self):
        return f"{self.cents // 100}.{self.cents % 100:02d}"


n = int(input())
amounts = [Money.parse(input().strip()) for _ in range(n)]
print(f"total {sum(amounts, Money(0))}")
print(f"max {max(amounts)}")
print("sorted", " ".join(str(m) for m in sorted(amounts)))
```

```
print(f"triple {3 * amounts[0]}")
print(f"distinct {len(set(amounts))}")
```

EXERCISE 2

A playlist that is a collection

Implement `Playlist` over an internal list of titles with `__len__`, `__contains__`, `__getitem__` (delegating to the list, so negative indexes and slices work), `__iter__` and `__bool__`. Run commands: `add title`, `len`, `has title`, `at i` (`no such track` on `IndexError`), `list` (all titles space-separated, or `empty` when the playlist is falsy).

Input: commands. **Output:** one line per query.

```
list
add intro
add outro
len
has intro
at -1
at 5
list
```

prints

```
empty
2
True
outro
no such track
intro outro
```

INPUT 1

```
list
add intro
add outro
len
has intro
at -1
at 5
list
```

OUTPUT 1

```
empty
2
True
outro
no such track
intro outro
```

REFERENCE SOLUTION

```
import sys


class Playlist:
    def __init__(self):
        self._tracks = []

    def add(self, title):
        self._tracks.append(title)

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

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

    def __getitem__(self, i):
        return self._tracks[i]

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

    def __bool__(self):
        return bool(self._tracks)


p = Playlist()
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "add":
        p.add(args[0])
    elif cmd == "len":
        print(len(p))
    elif cmd == "has":
        print(args[0] in p)
    elif cmd == "at":
        try:
            print(p[int(args[0])])
        except IndexError:
            print("no such track")
    elif cmd == "list":
        print(" ".join(p) if p else "empty")
```

QUIZ

1. What should `__eq__` return when `other` is of an unrelated type?

A. `False`

B. `NotImplemented`

C. `None`

D. Raise `TypeError`

2. Which method makes `print(x)` show custom text when only one is defined?

A. `__str__` only

B. `__repr__` — `str` falls back to it

C. `__format__`

D. `__print__`

3. How does `3 * money` reach your class?

A. Through `Money.__mul__`

B. `int.__mul__` returns `NotImplemented`, so Python calls `Money.__rmul__(money, 3)`

C. It cannot; only `money * 3` works

D. Through `__imul__`

4. What does `@functools.total_ordering` require and provide?

A. Requires `__lt__` (or another ordering method) and `__eq__`; provides the rest of the comparisons

B. Requires all six comparisons

C. Provides `__eq__` and `__hash__`

D. Requires `__cmp__`

5. A class defines `__len__` returning 0 for an empty collection and no `__bool__`. What is `bool(instance)` when empty?

A. `True`

B. `False`

C. `TypeError`

D. `None`

ANSWER KEY · DUNDER METHODS — MAKING A CLASS BEHAVE LIKE A BUILT-IN

1. B `NotImplemented` lets Python try the reflected operation and fall back to identity; returning `False` short-circuits that and can make `a == b` and `b == a` disagree.

2. B `print` uses `str`, which falls back to `__repr__` when `__str__` is absent. Defining only `__str__` leaves containers showing the default repr.

3. B The reflected method handles the case where the left operand does not know the right one.

4. A Define one ordering dunder plus `__eq__`; the decorator derives `__le__`, `__gt__`, `__ge__`.

5. B Truth testing falls back on `__len__`; zero length is falsy. Define `__bool__` to override.

LESSON 3 · 13 MIN

Properties and encapsulation — the underscore, @property and __slots__

Python has no `private` keyword and does not want one. Encapsulation is done by convention — a leading underscore means "not part of the interface" — and enforced socially rather than by the compiler, on the principle that consenting adults may reach inside an object if they accept the consequences. What Python does offer is `@property`, which lets an attribute *look* like plain data while being computed, validated or read-only, so that a class can start with public attributes and add control later without changing a single caller. This lesson covers the naming conventions and what name mangling actually does, `@property` with setters and deleters, computed and cached attributes, read-only attributes, and `__slots__`.

The underscore conventions

Name	Meaning
`balance`	public: part of the interface, safe to use
`_balance`	internal: use at your own risk; not imported by `from m import *`
`__balance`	name-mangled: stored as `_ClassName__balance` to avoid clashes in subclasses
`__init__`	a dunder: a protocol method Python calls; never invent your own

`_name` is the working convention and the only one most code needs. `__name` (two leading, no trailing underscores) is rewritten by the compiler to `_ClassName__name`; it is not privacy — `obj._Account__balance` reads it — but a way to keep a subclass's attribute of the same name from colliding with the base class's. Reserve it for that case.

Start with plain attributes

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
```

Java-style getters and setters (`get_radius`, `set_radius`) have no place in Python, because a public attribute can be turned into a property later with no change to callers. Write the plain attribute; add control when there is a reason.

@property

```
class Circle:
    def __init__(self, radius):
        self.radius = radius              # goes through the setter below

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("radius must be non-negative")
        self._radius = value

    @property
    def area(self):                       # computed; read-only because there
        return 3.141592653589793 * self._radius ** 2

c = Circle(2)
c.radius = 3          # setter runs: validated
c.area                # 28.27… — an attribute in syntax, a method in fact
c.area = 1            # AttributeError: can't set attribute
```

`@property` turns a method into an attribute read; `@name.setter` adds the write; `@name.deleter` handles `del c.radius`. The stored value lives in `_radius`; the public name `radius` is the property. Note that `__init__` assigns through the property, so validation applies from the start. Three uses: **validation** (reject bad values), **computed attributes** (`area` from `radius`, kept consistent automatically), and **read-only** attributes (a property with no setter).

A property should be cheap and side-effect-free — a reader sees `c.area` and assumes a field read. Anything expensive or that changes state is a method (`c.compute_area()`), so the call is visible.

Cached computed attributes

`functools.cached_property` computes once on first access and stores the result in the instance dict:

```
from functools import cached_property

class Report:
    def __init__(self, rows):
        self.rows = rows

    @cached_property
    def total(self):
        return sum(r.amount for r in self.rows)
```

Right for values that are expensive and do not change after construction; wrong if `rows` can be mutated later, because the cached value would be stale. `del report.total` clears the cache.

Read-only and immutable objects

A property without a setter gives one read-only attribute. For a wholly immutable object, the tools are a `frozen` dataclass (lesson 5), a `NamedTuple`, or a custom `__setattr__` that raises after `__init__` (Module 16). Immutable value objects are the easy ones to reason about, hash and share, and are what to reach for when a class represents a *value* (a point, a money amount, a date) rather than an *entity* with a life cycle (an account, a connection).

slots

By default every instance carries a `__dict__`. `__slots__` declares the attribute names up front, and the instance stores them in fixed slots instead:

```python
class Point:
    __slots__ = ("x", "y")

    def __init__(self, x, y):
        self.x, self.y = x, y

p = Point(1, 2)
p.z = 3               # AttributeError: 'Point' object has no attribute 'z'
```

Two effects: memory per instance drops (roughly by half for small objects, which matters for millions of points), and attribute typos become errors instead of silently creating new attributes. The cost: no dynamic attributes, and subclasses must declare their own `__slots__` or they get a `__dict__` back. Use it for small, numerous, fixed-shape objects; `@dataclass(slots=True)` writes it for you.

What encapsulation is for

The point is not secrecy but *invariants*: a class promises that `radius` is never negative, that `balance` equals the sum of its transactions, that `_index` always matches `_items`. Every write path that could break the promise — the constructor, a setter, a method — is where the check goes, and readers of the class can then assume the invariant everywhere else. An attribute that can be set to anything by anyone has no invariant to assume. Underscores mark the state that only the class's own methods should touch; properties guard the state that callers may set.

Pitfalls

- Java-style `get_x` / `set_x` methods.
- A property that does expensive work or has side effects.
- `__name` for privacy; it is for subclass collision avoidance.
- Assigning to `self.radius` in `__init__` *before* the property exists in a way that bypasses validation (it does go through the setter; but assigning to `self._radius` directly would not).
- `cached_property` on data that changes.
- A typo in an attribute name silently creating a new attribute — `__slots__` catches it.

Key takeaways

- `_name` marks internal state; `__name` is compiler-mangled to `_Class__name` for subclass safety; nothing is truly private.
- Start with plain attributes; `@property` and `@x.setter` add validation, computation or read-only-ness later without changing callers.
- Properties are cheap and pure; expensive work is a method; `cached_property` for expensive, stable values.
- Value objects should be immutable: frozen dataclass or `NamedTuple`.
- `__slots__` fixes the attribute set, saves memory and catches typos.

EXERCISE 1

Temperature with properties

Implement `Temperature` storing `_celsius`. `celsius` is a property whose setter rejects values below `-273.15` with `ValueError`; `fahrenheit` is a computed property with a setter that converts and stores through `celsius` (so validation applies to both). Commands: `c value` sets Celsius, `f value` sets Fahrenheit, `show` prints both with two decimals; an invalid value prints `invalid` and leaves the temperature unchanged.

Input: commands (the temperature starts at 0 °C). **Output:** one line per `show` and per rejected value.

```
c 100
show
f 32
show
c -300
show
```

prints

```
100.00C 212.00F
0.00C 32.00F
invalid
0.00C 32.00F
```

INPUT 1

```
c 100
show
f 32
show
c -300
show
```

OUTPUT 1

```
100.00C 212.00F
0.00C 32.00F
invalid
0.00C 32.00F
```

REFERENCE SOLUTION

```python
import sys


class Temperature:
    def __init__(self, celsius=0.0):
        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

    @property
    def fahrenheit(self):
        return self._celsius * 9 / 5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5 / 9


t = Temperature()
for line in sys.stdin:
    cmd, *args = line.split()
    try:
        if cmd == "c":
            t.celsius = float(args[0])
        elif cmd == "f":
            t.fahrenheit = float(args[0])
        elif cmd == "show":
            print(f"{t.celsius:.2f}C {t.fahrenheit:.2f}F")
    except ValueError:
        print("invalid")
```

EXERCISE 2

A slotted point

Define `Point` with `__slots__ = ("x", "y")`. Process commands: `set name value` assigns an attribute (an unknown name prints `AttributeError`), `get name` prints the value, and `dict` prints whether the instance has a `__dict__` (`hasattr(p, "__dict__")`).

Input: commands (the point starts at `0 0`). **Output:** one line per `get`, `dict` and failed `set`.

```
set x 5
set z 1
get x
dict
```

prints

```
AttributeError
5
False
```

INPUT 1

```
set x 5
set z 1
get x
dict
```

OUTPUT 1

```
AttributeError
5
False
```

REFERENCE SOLUTION

```python
import sys


class Point:
    __slots__ = ("x", "y")

    def __init__(self, x=0, y=0):
        self.x, self.y = x, y


p = Point()
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "set":
        try:
            setattr(p, args[0], int(args[1]))
        except AttributeError:
            print("AttributeError")
    elif cmd == "get":
        print(getattr(p, args[0]))
    elif cmd == "dict":
        print(hasattr(p, "__dict__"))
```

QUIZ

1. What does a single leading underscore, as in `_balance`, do?

A. Makes the attribute private to the class

B. Signals 'internal' by convention; nothing is enforced (and `import *` skips it)

C. Mangles the name

D. Makes it read-only

2. Where is `self.__secret` actually stored for a class `Vault`?

A. `__secret`

B. `_Vault__secret`

C. `_secret`

D. It is not stored — it is a compile-time constant

3. Why prefer a plain attribute over `get_x()` / `set_x()` methods?

A. Methods are slower

B. A plain attribute can later become a `@property` without changing any caller

C. Methods cannot be documented

D. Attributes are private

4. What happens on `c.area = 5` when `area` is a `@property` with no setter?

A. The value is stored in `__dict__`

B. `AttributeError: can't set attribute`

C. The getter is called with 5

D. It is silently ignored

5. Which is a consequence of `__slots__`?

A. Instances cannot have any attributes

B. Instances have no `__dict__`, use less memory, and reject attributes not named in the slots

C. Attributes become private

D. The class cannot be subclassed

ANSWER KEY · PROPERTIES AND ENCAPSULATION — THE UNDERSCORE, @PROPERTY AND __SLOTS__

1. B Python has no access control; the underscore is a message to readers. Double underscores add name mangling, still not privacy.

2. B Name mangling prefixes the class name so a subclass's `__secret` does not collide. `vault._Vault__secret` still reads it.

3. B Properties keep attribute syntax while adding validation or computation, so encapsulation can be introduced when needed rather than up front.

4. B A property without a setter is read-only; the assignment is refused.

5. B Slots reserve fixed storage for the named attributes; a typo in an attribute name becomes an `AttributeError` rather than a silent new attribute.

LESSON 4 · 12 MIN

Class methods and static methods

Not every function in a class body needs an instance. A method that builds an instance from some other form — a string, a dictionary, a file — has no `self` to work with yet; a helper that merely belongs with the class conceptually needs neither the instance nor the class. Python marks these with two decorators: `@classmethod` receives the class as its first argument, `@staticmethod` receives nothing. This lesson explains both, the alternative-constructor pattern that is `@classmethod`'s main job, why class methods respect subclassing where hard-coded class names do not, class-level counters and constants, and the case for a module-level function instead.

Three kinds of method

```
class Temperature:
    unit = "C"                                  # class attribute

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

    def describe(self):                         # instance method: needs the c
        return f"{self.degrees}°{self.unit}"

    @classmethod
    def from_fahrenheit(cls, f):                # class method: gets the class
        return cls((f - 32) * 5 / 9)

    @staticmethod
    def is_valid(degrees):                      # static method: gets nothing
        return degrees ⩾ -273.15
```

```
t = Temperature.from_fahrenheit(212)   # Temperature(100.0)
t.describe()                           # '100.0°C'
Temperature.is_valid(-300)             # False
t.is_valid(20)                         # True — callable through an instance t
```

The first parameter of a class method is the class itself, conventionally `cls`; the decorator arranges that whether the method is called on the class or on an instance. A static method is

a plain function that happens to live in the class namespace — it could be a module-level function, and often should be.

Alternative constructors

`__init__` takes one shape of arguments. When an object can be built from several, class methods named `from_…` are the idiom:

```
class Date:
    def __init__(self, year, month, day):
        self.year, self.month, self.day = year, month, day

    @classmethod
    def from_iso(cls, text):                    # "2024-05-01"
        y, m, d = map(int, text.split("-"))
        return cls(y, m, d)

    @classmethod
    def from_ordinal(cls, n):
        ...
        return cls(y, m, d)

    @classmethod
    def today(cls):
        import datetime
        t = datetime.date.today()
        return cls(t.year, t.month, t.day)
```

Each parses or computes, then calls `cls(...)` — not `Date(...)`. The difference matters the moment someone subclasses: `class UTCDate(Date)` inherits `from_iso`, and `UTCDate.from_iso("2024-05-01")` returns a `UTCDate` because `cls` *is* `UTCDate`. A hard-coded `Date(...)` would return the base class from a subclass's constructor. The standard library does this everywhere: `dict.fromkeys`, `datetime.fromtimestamp`, `int.from_bytes`, `Path.home()`.

Class-level state

A class method is also the natural way to read or change state that belongs to the class rather than to an instance:

```
class Account:
    _count = 0
    _rate = 0.02

    def __init__(self, owner):
        self.owner = owner
        type(self)._count += 1            # or Account._count; cls in a classm

    @classmethod
    def count(cls):
        return cls._count

    @classmethod
    def set_rate(cls, rate):
        cls._rate = rate
```

`type(self)` inside an instance method is the object's actual class — the same idea as `cls`. Writing `self._count += 1` would be the class-attribute trap from lesson 1: it reads the class value and then creates an *instance* attribute, leaving the class counter untouched.

Class-level registries, caches and configuration follow the same pattern, and so does the singleton — one shared instance returned by a class method — which is usually better replaced by a module-level object (Module 12), since a module is already a singleton.

Static methods: when, and when not

A static method is right when a function is *about* the class — a validation rule, a conversion helper, a factory of related values — and grouping it under the class name helps the reader find it, but it uses neither instance nor class state:

```
class Matrix:
    @staticmethod
    def identity_rows(n):
        return [[int(i == j) for j in range(n)] for i in range(n)]
```

The alternative is a module-level function next to the class, which is what most Python code does: shorter to call, no decorator, trivially testable. Reach for `@staticmethod` when the function would otherwise be an orphan, when subclasses may want to override it, or when a name like `Matrix.identity_rows` documents the relationship. Never use it for something that actually needs `cls` — that is a class method.

How the decorators work

`@classmethod` and `@staticmethod` wrap the function in a *descriptor* that changes what attribute lookup hands back: a bound method with `cls` filled in, or the bare function (Module 16). That is why they can be called on either the class or an instance, and why a plain function in the class body — no decorator — becomes an instance method expecting `self`. Calling `Temperature.describe()` without an instance is therefore the familiar `TypeError: missing 1 required positional argument: 'self'`.

Pitfalls

`return Date(...)` instead of `return cls(...)` in a class method — subclasses get the wrong type.

`self._count += 1` for a class-level counter — it creates an instance attribute.

`@staticmethod` on a function that uses `cls` or `self`.

A `@classmethod` where an instance's data is needed.

Calling an instance method on the class without an instance.

A class used only as a namespace for static methods — that is a module.

Key takeaways

Instance methods get `self`, class methods get `cls` (via `@classmethod`), static methods get nothing (via `@staticmethod`).

`from_…` class methods are alternative constructors; they call `cls(...)` so subclasses get instances of themselves.

Class-level counters and settings are read and changed through `cls` or `type(self)`, never `self.x += …`.

A static method is a namespaced helper; a module-level function is usually the simpler choice.

The decorators are descriptors that decide what `obj.method` returns; an undecorated function is an instance method.

EXERCISE 1

Durations from several forms

Implement `Duration` holding total seconds with two alternative constructors: `Duration.from_text("1h30m15s")` (each of `h`, `m`, `s` optional, in that order) and `Duration.from_seconds(n)` — both must build the instance through `cls(...)`. `__str__` renders `1h30m15s` always showing all three parts, and `__add__` adds two durations. Commands: `text <spec>`, `seconds <n>` (each prints the parsed duration and its total seconds), `sum` (prints the sum of every duration parsed so far).

Input: commands. **Output:** `<h>h<m>m<s>s = <total>` per parse, and `<h>h<m>m<s>s` per `sum`.

```
text 1h30m
seconds 5400
sum
```

prints

```
1h30m0s = 5400
1h30m0s = 5400
3h0m0s
```

INPUT 1

```
text 1h30m
seconds 5400
sum
```

OUTPUT 1

```
1h30m0s = 5400
1h30m0s = 5400
3h0m0s
```

REFERENCE SOLUTION

```
import re
import sys


class Duration:
    PATTERN = re.compile(r"(?:(\d+)h)?(?:(\d+)m)?(?:(\d+)s)?")

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

    @classmethod
    def from_text(cls, text):
        h, m, s = (int(x) if x else 0 for x in cls.PATTERN.fullmatch(text).groups())
        return cls(h * 3600 + m * 60 + s)

    @classmethod
    def from_seconds(cls, n):
        return cls(int(n))

    def __add__(self, other):
        if not isinstance(other, Duration):
            return NotImplemented
        return Duration(self.seconds + other.seconds)

    def __str__(self):
        h, rest = divmod(self.seconds, 3600)
        m, s = divmod(rest, 60)
        return f"{h}h{m}m{s}s"


parsed = []
for line in sys.stdin:
    cmd, *rest = line.split()
    if cmd == "sum":
        print(sum(parsed, Duration(0)))
        continue
    d = Duration.from_text(rest[0]) if cmd == "text" else Duration.from_seconds(rest
    parsed.append(d)
    print(f"{d} = {d.seconds}")
```

EXERCISE 2

Usernames with a static validator

Implement `Username` with a `@staticmethod is_valid(text)` — 3 to 12 characters, letters, digits and underscores only, starting with a letter — that the constructor calls, raising `ValueError` for an invalid

name, and a class attribute `created` counting successful constructions (increment via `type(self)`). Commands: `check text` prints `valid` / `invalid` using the static method directly; `create text` prints `created <name>` or `rejected`; `count` prints the class count.

Input: commands. **Output:** one line per command.

```
check ada_1
check 1ada
create ada_1
create x
count
```

prints

```
valid
invalid
created ada_1
rejected
1
```

INPUT 1

```
check ada_1
check 1ada
create ada_1
create x
count
```

OUTPUT 1

```
valid
invalid
created ada_1
rejected
1
```

REFERENCE SOLUTION

```
import re
import sys


class Username:
    created = 0
    RULE = re.compile(r"[A-Za-z][A-Za-z0-9_]{2,11}")

    def __init__(self, text):
        if not self.is_valid(text):
            raise ValueError(text)
        self.text = text
        type(self).created += 1

    @staticmethod
    def is_valid(text):
        return Username.RULE.fullmatch(text) is not None


for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "check":
        print("valid" if Username.is_valid(args[0]) else "invalid")
    elif cmd == "create":
        try:
            u = Username(args[0])
            print(f"created {u.text}")
        except ValueError:
            print("rejected")
    elif cmd == "count":
        print(Username.created)
```

QUIZ

1. What does a `@classmethod` receive as its first argument?

A. The instance

B. The class it was called on (or the class of the instance it was called on)

C. Nothing

D. The module

2. Why write `return cls(y, m, d)` rather than `return Date(y, m, d)` in `Date.from_iso`?

A. `cls` is faster

B. So that `SubDate.from_iso(...)` returns a `SubDate`

C. `Date` is not defined inside its own methods

D. There is no difference

3. What does `self.count += 1` do when `count` is a class attribute?

A. Increments the shared counter

B. Creates an instance attribute `count` that shadows the class one; the class value is unchanged

C. Raises `AttributeError`

D. Increments both

4. When is `@staticmethod` the right choice?

A. For any method that modifies the instance

B. For a helper that uses neither instance nor class state but belongs with the class conceptually

C. For alternative constructors

D. For every method in a dataclass

5. What does calling `Temperature.describe()` (an instance method) without an instance do?

A. Works, with `self` as `None`

B. `TypeError: missing 1 required positional argument: 'self'`

C. Returns a bound method

D. Creates an instance automatically

ANSWER KEY · CLASS METHODS AND STATIC METHODS

1. B `cls` is the class — which, for a subclass, is the subclass. That is what makes `cls(...)` in a factory return the right type.

2. B Hard-coding the class name makes subclasses' inherited constructors return the base class.

3. B The read finds the class attribute, but the assignment binds on the instance. Use `type(self).count += 1` or `ClassName.count += 1`.

4. B Static methods are namespaced functions; constructors need `cls`, and instance behaviour needs `self`.

5. B Accessed on the class, an instance method is a plain function that still expects its first argument.

LESSON 5 · 14 MIN

Dataclasses — classes that are mostly data

Most classes are records: a few named fields, an `__init__` that stores them, a `__repr__` that shows them, an `__eq__` that compares them. Writing those by hand is twenty lines of boilerplate per class and a bug every time a field is added to one method and not the others. `@dataclass` (3.7) generates them from the field annotations, and its options add ordering, immutability, hashing, slots and keyword-only construction. This lesson covers the decorator, defaults and `field`, the generated methods, the options that matter, `__post_init__` for validation and derived fields, the helpers `asdict` / `astuple` / `replace`, and when a plain class is still the better tool.

The basic form

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float = 0.0

p = Point(1.5)
p                  # Point(x=1.5, y=0.0)   — __repr__
p == Point(1.5, 0) # True                  — __eq__ compares fields as a tuple
p.x = 2            # mutable by default
```

Each annotated class attribute becomes a field, in declaration order; the annotations are the field types (not enforced, like all hints). The decorator writes `__init__(self, x, y=0.0)`, `__repr__` and `__eq__`. Fields with defaults must come after those without, as in any signature.

Defaults and field()

```python
from dataclasses import dataclass, field

@dataclass
class Team:
    name: str
    members: list[str] = field(default_factory=list)    # a fresh list per ins
    size_limit: int = 10
    _id: int = field(default=0, repr=False, compare=False)
```

A mutable default (`members: list = []`) is refused with `ValueError: mutable default … use default_factory` — the dataclass machinery knows the trap from Module 4. `default_factory` takes a zero-argument callable, called per instance. `field()` also controls whether a field appears in `repr`, `eq`/ordering (`compare=False`), `__init__` (`init=False` for a field computed later), and carries `metadata`.

Options

```python
@dataclass(order=True)          # __lt__, __le__, __gt__, __ge__ comparing fie
@dataclass(frozen=True)         # assignment after __init__ raises; instances
@dataclass(slots=True)          # __slots__ generated (3.10): smaller, no dyn
@dataclass(kw_only=True)        # every field keyword-only in __init__ (3.10)
@dataclass(eq=False)            # keep identity comparison and the default has
```

`order=True` compares instances as tuples of their fields, which is right when the declaration order is the sort order (a `Version(major, minor)`); when it is not, define `__lt__` yourself or put a sort key first. `frozen=True` gives a value object: `p.x = 2` raises `FrozenInstanceError`, and because the fields cannot change, the decorator generates `__hash__` so instances work as dict keys and set members. `frozen=True, slots=True` together is the tightest value type Python offers without writing dunders. `kw_only=True` prevents positional-argument mix-ups in classes with many same-typed fields.

Hashing follows the rules of Module 7: an `eq=True` (the default), non-frozen dataclass is unhashable; `unsafe_hash=True` forces a hash on a mutable class, which is exactly as unsafe as it sounds.

---

post_init

`__init__` is generated, so validation and derived fields go in `__post_init__`, which runs right after it:

```python
@dataclass
class Rect:
    width: float
    height: float
    area: float = field(init=False)

    def __post_init__(self):
        if self.width < 0 or self.height < 0:
            raise ValueError("negative dimension")
        self.area = self.width * self.height
```

In a frozen dataclass a derived field is set with `object.__setattr__(self, "area", …)` inside `__post_init__`, because normal assignment is blocked. `InitVar[T]` declares a pseudo-field that is passed to `__init__` and `__post_init__` but not stored.

Helpers

```python
from dataclasses import asdict, astuple, replace, fields

asdict(p)                       # {'x': 1.5, 'y': 0.0} — recursive on nested da
astuple(p)                      # (1.5, 0.0)
replace(p, y=3)                 # a new Point(x=1.5, y=3) — the frozen-friendly
[f.name for f in fields(Point)]   # ['x', 'y']
```

`asdict` is the bridge to JSON (`json.dumps(asdict(p))`); `replace` is how a frozen instance is "changed". Dataclasses are ordinary classes: methods, properties, class methods (`from_dict`) and inheritance all work, and a subclass's fields are appended after the base's.

When not to use a dataclass

The class has significant **behaviour and invariants** that `__init__` should enforce in a specific order — a hand-written `__init__` is clearer than `__post_init__` gymnastics.

Equality should be **identity** (an account, a connection): use `eq=False` or a plain class.

You need a `NamedTuple`: same field syntax, always immutable, unpackable, slightly lighter, but it *is* a tuple (`Point(1, 2) == (1, 2)` is `True`, which is rarely wanted).

A plain **dict** is the right shape for loosely structured data that is never operated on.

The rule of thumb: a `@dataclass` for records with a fixed shape, `frozen=True` when they are values, a plain class when the constructor has logic, `NamedTuple` when tuple behaviour is wanted.

Pitfalls

A mutable default without `default_factory` (refused) — or a *shared object* passed as `default=` (accepted, and shared).

`order=True` when the field order is not the sort order.

Expecting type annotations to be checked.

Forgetting that `asdict` copies deeply — large nested structures get duplicated.

Assigning in `__post_init__` of a frozen class without `object.__setattr__`.

A non-frozen dataclass used as a dict key (`TypeError`).

Key takeaways

`@dataclass` generates `__init__`, `__repr__` and `__eq__` from annotated fields; `field(default_factory=…)` for mutable defaults.

`order=True` for tuple-style ordering, `frozen=True` for immutable, hashable value objects, `slots=True` for compact instances, `kw_only=True` for safe construction.

`__post_init__` validates and derives; `InitVar` passes construction-only arguments.

`asdict`, `astuple`, `replace`, `fields` are the helpers; `replace` is how frozen instances change.

Use a plain class when the constructor has logic or equality is identity; `NamedTuple` when you want a tuple.

EXERCISE 1

Employee records

Define `@dataclass(order=True) Employee` with fields `dept: str`, `salary: int`, `name: str` — in that order, so the generated ordering sorts by department, then salary, then name. Read `n` records as `name dept salary`, print them sorted with plain `sorted()`, then print `asdict` of the first sorted record as JSON with sorted keys, then a copy of it made with `replace(salary=...)` doubled.

Input: `n`, then `n` lines. **Output:** `n` lines `Employee(dept='eng', salary=100, name='ada')` (the generated repr), then the JSON line, then the replaced record's repr.

```
2
bob ops 90
ada eng 100
```

prints

```
Employee(dept='eng', salary=100, name='ada')
Employee(dept='ops', salary=90, name='bob')
{"dept": "eng", "name": "ada", "salary": 100}
Employee(dept='eng', salary=200, name='ada')
```

INPUT 1

```
2
bob ops 90
ada eng 100
```

OUTPUT 1

```
Employee(dept='eng', salary=100,
name='ada')
Employee(dept='ops', salary=90,
name='bob')
{"dept": "eng", "name": "ada", "salary":
100}
Employee(dept='eng', salary=200,
name='ada')
```

REFERENCE SOLUTION

```
import json
from dataclasses import dataclass, asdict, replace


@dataclass(order=True)
class Employee:
    dept: str
    salary: int
    name: str


n = int(input())
staff = []
for _ in range(n):
    name, dept, salary = input().split()
    staff.append(Employee(dept, int(salary), name))
ordered = sorted(staff)
for e in ordered:
    print(e)
first = ordered[0]
print(json.dumps(asdict(first), sort_keys=True))
print(replace(first, salary=first.salary * 2))
```

EXERCISE 2

Frozen points

Define `@dataclass(frozen=True) Point` with integer `x` and `y` and a `__post_init__` that raises `ValueError` when either is negative. Read `n` lines `x y`; for each print `ok` or `invalid`. Then print the number of distinct valid points (they are hashable), and try to assign `x = 0` on the first valid point — print `frozen` when `FrozenInstanceError` is raised — and finally print `replace(first, x=0)`.

Input: `n`, then `n` lines. **Output:** `n` lines, then `distinct <k>`, `frozen`, and the replaced repr (the last two only if there is a valid point).

```
3
1 2
-1 0
1 2
```

prints

```
ok
invalid
ok
distinct 1
frozen
Point(x=0, y=2)
```

INPUT 1

```
3
1 2
-1 0
1 2
```

OUTPUT 1

```
ok
invalid
ok
distinct 1
frozen
Point(x=0, y=2)
```

REFERENCE SOLUTION

```python
from dataclasses import dataclass, replace, FrozenInstanceError


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

    def __post_init__(self):
        if self.x < 0 or self.y < 0:
            raise ValueError("negative coordinate")


n = int(input())
points = []
for _ in range(n):
    x, y = map(int, input().split())
    try:
        points.append(Point(x, y))
        print("ok")
    except ValueError:
        print("invalid")
print(f"distinct {len(set(points))}")
if points:
    first = points[0]
    try:
        first.x = 0
    except FrozenInstanceError:
        print("frozen")
    print(replace(first, x=0))
```

QUIZ

1. Which methods does a plain `@dataclass` generate?

A. `__init__`, `__repr__`, `__eq__`

B. `__init__` only

C. `__init__`, `__repr__`, `__eq__`, `__lt__`, `__hash__`

D. `__repr__` and `__str__`

2. What happens with `tags: list = []` as a dataclass field?

A. Every instance shares one list

B. `ValueError: mutable default … use default_factory`

C. Each instance gets a new list

D. `SyntaxError`

3. What does `frozen=True` change?

A. Fields cannot be reassigned after `__init__`, and instances become hashable

B. Fields are private

C. The class cannot be subclassed

D. Only `__repr__` is generated

4. Where does validation belong in a dataclass?

A. In `__init__` — override it

B. In `__post_init__`, which runs after the generated `__init__`

C. In `__repr__`

D. Dataclasses cannot validate

5. What does `order=True` compare?

A. The `repr` strings

B. The fields as a tuple, in declaration order

C. The `id`s

D. Only the first field

ANSWER KEY · DATACLASSES — CLASSES THAT ARE MOSTLY DATA

1. A Ordering needs `order=True`; hashing needs `frozen=True` (or `unsafe_hash`).

2. B The decorator refuses mutable defaults outright; `field(default_factory=list)` is the fix.

3. A Immutability lets the decorator safely generate `__hash__` from the fields; `replace()` is how a 'modified' copy is made.

4. B `__init__` is generated; `__post_init__` is the hook for checks and derived fields.

5. B Instances compare like tuples of their fields, so declaration order is the sort order — declare the sort key first.

LESSON 6 · 14 MIN

Designing a class — invariants, interfaces and a worked example

The mechanics of classes are the previous five lessons; this one is about judgement. A class earns its existence by protecting an *invariant* — a fact about its data that every method maintains and every caller may assume — and by presenting a *small interface* that says what the object does rather than how it stores things. This lesson works through the design of one class, `Inventory`, from the questions to ask before writing it, through representation, constructor, invariants and interface, to the tests that pin it, and ends with the checklist that separates a class that helps from a class that merely groups functions.

The questions before the code

What is the invariant? For an inventory: quantities are never negative; every listed item has a name and a price; the total value equals the sum over items.

Is it a value or an entity? A `Money` is a value — equal if the amounts are equal, immutable, hashable. An `Inventory` is an entity — it has a life cycle, is mutated, is equal only to itself.

What must callers be able to do? Add stock, remove stock (and be told if there is not enough), ask what is there, ask the total value. That is the interface; everything else is internal.

What must callers *not* do? Set a quantity negative, corrupt the total, depend on the storage layout.

If there is no invariant and no behaviour — just fields — the answer is a dataclass, or a dict, not a class with methods.

Representation

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Item:                          # a value: what an item *is*
    name: str
    price_cents: int

class Inventory:                     # an entity: what the stock *does*
    def __init__(self):
        self._stock: dict[str, tuple[Item, int]] = {}   # name → (item, quant
```

The storage is a dict keyed by name because lookups by name are what the operations need; it is prefixed with an underscore because no caller should touch it; the value type is a tuple of the item and its count. Prices are integer cents (Module 2). The representation is chosen for the *operations*, and it can change later without changing the interface.

Constructor and invariants

```python
    def add(self, item: Item, quantity: int = 1) -> None:
        if quantity <= 0:
            raise ValueError(f"quantity must be positive, got {quantity}")
        _, current = self._stock.get(item.name, (item, 0))
        self._stock[item.name] = (item, current + quantity)

    def remove(self, name: str, quantity: int = 1) -> None:
        if name not in self._stock:
            raise KeyError(name)
        item, current = self._stock[name]
        if quantity > current:
            raise ValueError(f"only {current} {name} in stock, cannot remove 
        if current == quantity:
            del self._stock[name]
        else:
            self._stock[name] = (item, current - quantity)
```

Every path that changes `_stock` checks the invariant before changing anything, so a failed call leaves the inventory exactly as it was — the *strong guarantee*, which is easy here because each method makes one assignment after all its checks. The exceptions are the built-in ones with the right meaning (`ValueError` for a bad value, `KeyError` for a missing

key) and carry messages a caller can act on; Module 10 discusses custom exceptions for when these are not specific enough.

Interface

```
    def quantity(self, name: str) -> int:
        item_and_count = self._stock.get(name)
        return item_and_count[1] if item_and_count else 0

    def total_value_cents(self) -> int:
        return sum(item.price_cents * n for item, n in self._stock.values())

    def __len__(self) -> int:              # number of distinct items
        return len(self._stock)

    def __contains__(self, name: str) -> bool:
        return name in self._stock

    def __iter__(self):                    # (item, quantity) pairs, by name
        for name in sorted(self._stock):
            yield self._stock[name]

    def __repr__(self) -> str:
        return f"Inventory({len(self)} items, {self.total_value_cents()} cents
```

Queries are read-only methods with clear names; the dunders make the object usable with `len`, `in` and `for` (Module 8 lesson 2). Note what is *not* there: no `get_stock()` returning the dict (that would hand the invariant to the caller), no `set_quantity` (the operations are add and remove, which is what the domain has). Iteration sorts by name so that output is deterministic. A `report()` that prints would be a mistake — printing is the caller's job; the class returns data.

Tests as the specification

```
inv = Inventory()
bolt = Item("bolt", 5)
inv.add(bolt, 100)
inv.remove("bolt", 30)
assert inv.quantity("bolt") == 70
assert inv.total_value_cents() == 350
try:
    inv.remove("bolt", 1000)
except ValueError:
    pass
assert inv.quantity("bolt") == 70          # unchanged after the failed call
assert "bolt" in inv and len(inv) == 1
```

Each test is an invariant made executable: the failed removal did not change anything; the count and the value agree with the operations. Module 18 turns these into a real test suite; the point here is that the tests were written *from the invariants*, and a class whose invariants you cannot state is a class you cannot test.

The checklist

- [ ] The class has a one-sentence purpose and an invariant you can write down.
- [ ] `__init__` establishes the invariant; every mutating method checks before it changes.
- [ ] Internal state has underscores; the interface is a handful of verbs (mutators) and nouns (queries).
- [ ] `__repr__` always; `__eq__` / `__hash__` for values, not for entities; the collection dunders when the object *is* a collection.
- [ ] Methods return data, never print; I/O stays in `main`.
- [ ] Value types are frozen dataclasses; an object with no behaviour is a dataclass or a dict.
- [ ] Composition over inheritance by default (Module 9): an `Inventory` *has* a dict, it is not one.

Pitfalls

- A class that is a bag of loosely related functions with no shared state.
- Exposing the internal collection through a getter.
- Mutating state before validating, so a failed call leaves a half-updated object.
- Methods that print, making the class unusable in any other context.

Inheriting from `dict` or `list` to "reuse" their methods — you inherit forty methods that can break the invariant.

Naming by mechanism (`process`, `handle`, `data`) instead of by meaning (`remove`, `total_value_cents`).

Key takeaways

A class protects an invariant and offers a small interface; without an invariant, use a dataclass or a dict.

Decide value versus entity first: values are frozen and compare by fields; entities are mutable and compare by identity.

Check before changing, so a failed call changes nothing; raise the built-in exception with the right meaning.

Hide the representation, expose verbs and queries, add the collection dunders when they fit, return data rather than printing.

Write the tests from the invariants — they are the specification.

EXERCISE 1

Inventory

Implement the lesson's `Inventory`: `add(name, price_cents, qty)` (rejects `qty ⩽ 0` with `ValueError`; a repeated name adds to the quantity and keeps the first price), `remove(name, qty)` (`KeyError` for an unknown name, `ValueError` when there is not enough), `quantity(name)`, `total_value_cents()`, `__len__`, `__contains__` and `__iter__` yielding `(name, price_cents, qty)` sorted by name. Commands: `add name price qty`, `remove name qty`, `qty name`, `value`, `list`; errors print `unknown item`, `bad quantity` or `not enough`.

Input: commands. **Output:** one line per query and per error; `list` prints `name qty price` per item or `empty`.

```
add bolt 5 100
add nut 2 250
remove bolt 30
remove nut 999
remove screw 1
qty bolt
value
list
```

prints

```
not enough
unknown item
70
850
bolt 70 5
nut 250 2
```

INPUT 1

```
add bolt 5 100
add nut 2 250
remove bolt 30
remove nut 999
remove screw 1
qty bolt
value
list
```

OUTPUT 1

```
not enough
unknown item
70
850
bolt 70 5
nut 250 2
```

REFERENCE SOLUTION

```
import sys


class Inventory:
    def __init__(self):
        self._stock = {}  # name -> [price_cents, qty]

    def add(self, name, price_cents, qty):
        if qty <= 0:
            raise ValueError("bad quantity")
        if name in self._stock:
            self._stock[name][1] += qty
        else:
            self._stock[name] = [price_cents, qty]

    def remove(self, name, qty):
        if name not in self._stock:
            raise KeyError(name)
        if qty > self._stock[name][1]:
            raise ValueError("not enough")
        self._stock[name][1] -= qty
        if self._stock[name][1] == 0:
            del self._stock[name]

    def quantity(self, name):
        return self._stock[name][1] if name in self._stock else 0

    def total_value_cents(self):
        return sum(price * qty for price, qty in self._stock.values())

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

    def __contains__(self, name):
        return name in self._stock

    def __iter__(self):
        for name in sorted(self._stock):
            price, qty = self._stock[name]
            yield name, price, qty


inv = Inventory()
for line in sys.stdin:
    cmd, *args = line.split()
    try:
        if cmd == "add":
            inv.add(args[0], int(args[1]), int(args[2]))
        elif cmd == "remove":
```

```
            inv.remove(args[0], int(args[1]))
        elif cmd == "qty":
            print(inv.quantity(args[0]))
        elif cmd == "value":
            print(inv.total_value_cents())
        elif cmd == "list":
            if inv:
                for name, price, qty in inv:
                    print(name, qty, price)
            else:
                print("empty")
    except KeyError:
        print("unknown item")
    except ValueError as e:
        print(e)
```

EXERCISE 2

A bounded stack

Implement `BoundedStack(capacity)` whose invariant is `0 ≤ len(stack) ≤ capacity`: `push(x)` raises `OverflowError` when full, `pop()` and `peek()` raise `IndexError` when empty, `__len__` and `__repr__` (`BoundedStack([1, 2], capacity=3)`). Commands: `push x`, `pop`, `peek`, `len`, `show`; errors print `full` or `empty`.

Input: the capacity, then commands. **Output:** one line per `pop`, `peek`, `len`, `show` and per error.

```
2
push 1
push 2
push 3
pop
show
```

prints

```
full
2
BoundedStack([1], capacity=2)
```

INPUT 1

```
2
push 1
push 2
push 3
pop
show
```

OUTPUT 1

```
full
2
BoundedStack([1], capacity=2)
```

REFERENCE SOLUTION

```
import sys


class BoundedStack:
    def __init__(self, capacity):
        self.capacity = capacity
        self._items = []

    def push(self, x):
        if len(self._items) ⩾ self.capacity:
            raise OverflowError("full")
        self._items.append(x)

    def pop(self):
        if not self._items:
            raise IndexError("empty")
        return self._items.pop()

    def peek(self):
        if not self._items:
            raise IndexError("empty")
        return self._items[-1]

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

    def __repr__(self):
        return f"BoundedStack({self._items!r}, capacity={self.capacity})"


stack = BoundedStack(int(input()))
for line in sys.stdin:
    cmd, *args = line.split()
    try:
        if cmd == "push":
            stack.push(int(args[0]))
        elif cmd == "pop":
            print(stack.pop())
        elif cmd == "peek":
            print(stack.peek())
        elif cmd == "len":
            print(len(stack))
        elif cmd == "show":
            print(stack)
    except OverflowError:
        print("full")
    except IndexError:
        print("empty")
```

QUIZ

1. What justifies writing a class rather than a dict or a dataclass?

A. Having more than three fields

B. An invariant to protect and behaviour that maintains it, behind a small interface

C. Needing a `__repr__`

D. Needing to sort instances

2. Why check arguments *before* mutating state in a method?

A. It is faster

B. So a failed call leaves the object unchanged — the strong guarantee

C. Python requires it

D. To avoid `__repr__` being called

3. Which is the design smell?

A. A method that returns a computed value

B. A `get_stock()` method returning the internal dict

C. A `__repr__` that shows the state

D. A frozen dataclass for a value

4. Why should a class's methods not `print`?

A. Printing is slow

B. The class becomes unusable anywhere output is not wanted; return data and let the caller print

C. `print` cannot be called from methods

D. It breaks `__repr__`

5. Which pair is 'value versus entity'?

A. `int` versus `float`

B. A `Money` amount versus a bank `Account`

C. A list versus a tuple

D. A class versus an instance

ANSWER KEY · DESIGNING A CLASS — INVARIANTS, INTERFACES AND A WORKED EXAMPLE

1. B Fields alone are a dataclass or a dict; a class with methods earns its place by guarding a fact about its data.

2. B Validating first means an exception cannot leave a half-updated object behind.

3. B Handing out the internal collection lets any caller break the invariant; expose operations and queries instead.

4. B I/O belongs at the edge (`main`); a method that returns values can be tested, composed and reused.

5. B Values compare by their fields and are immutable; entities have a life cycle, change, and are equal only to themselves.

CHECKPOINT · 25 MIN

Checkpoint — Classes and objects

This checkpoint covers the whole module: the `class` statement with `__init__`, `self`, instance and class attributes, the dunder methods that make a class behave like a built-in, `@property` and the encapsulation conventions with `__slots__`, class methods as alternative constructors and static methods, dataclasses with their options and `__post_init__`, and the design discipline of invariants and small interfaces.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What does `obj.method(x)` become, and why is `self` the first parameter?
- Why is `members = []` in a class body a bug, and where does per-instance state belong?
- What must `__eq__` return for a foreign type, and what happens to `__hash__` when `__eq__` is defined?
- Which dunders make `len(x)`, `x in c`, `for e in c` and `3 * x` work?
- Why does a `from_…` class method call `cls(...)` rather than the class by name?
- What does `@dataclass(frozen=True)` give you that the default does not?
- Where does validation go in a dataclass, and in a hand-written class?

The three programs are a `Fraction` value type with arithmetic, comparison and hashing dunders, a bank ledger whose accounts enforce invariants and are built through class-method constructors, and a frozen dataclass catalogue with derived fields, sorting and `replace`.

EXERCISE 1

A fraction value type

Implement `Frac(num, den)` normalised in `__init__`: reduce by the gcd and keep the sign in the numerator (a zero denominator raises `ValueError`). Define `__add__`, `__sub__`, `__mul__`, `__truediv__` (division by a zero fraction raises `ZeroDivisionError`), `__eq__`, `__lt__`, `__hash__` and `__str__` (`3/4`, or just `2` when the denominator is 1). Evaluate lines `a/b op c/d` and print the result; print the exception name for a failure.

Input: lines. **Output:** one line per input line.

```
1/2 + 1/3
3/4 * 4/3
1/2 - 1/2
1/3 / 0/5
```

prints

```
5/6
1
0
ZeroDivisionError
```

INPUT 1

```
1/2 + 1/3
3/4 * 4/3
1/2 - 1/2
1/3 / 0/5
```

OUTPUT 1

```
5/6
1
0
ZeroDivisionError
```

REFERENCE SOLUTION

```python
import math
import sys


class Frac:
    def __init__(self, num, den=1):
        if den == 0:
            raise ValueError("zero denominator")
        if den < 0:
            num, den = -num, -den
        g = math.gcd(num, den) or 1
        self.num, self.den = num // g, den // g

    def __add__(self, o):
        return Frac(self.num * o.den + o.num * self.den, self.den * o.den)

    def __sub__(self, o):
        return Frac(self.num * o.den - o.num * self.den, self.den * o.den)

    def __mul__(self, o):
        return Frac(self.num * o.num, self.den * o.den)

    def __truediv__(self, o):
        if o.num == 0:
            raise ZeroDivisionError("division by zero fraction")
        return Frac(self.num * o.den, self.den * o.num)

    def __eq__(self, o):
        if not isinstance(o, Frac):
            return NotImplemented
        return (self.num, self.den) == (o.num, o.den)

    def __lt__(self, o):
        return self.num * o.den < o.num * self.den

    def __hash__(self):
        return hash((self.num, self.den))

    def __str__(self):
        return f"{self.num}" if self.den == 1 else f"{self.num}/{self.den}"


OPS = {"+": "__add__", "-": "__sub__", "*": "__mul__", "/": "__truediv__"}
for line in sys.stdin:
    a, op, b = line.split()
    try:
        x = Frac(*map(int, a.split("/")))
        y = Frac(*map(int, b.split("/")))
        print(getattr(x, OPS[op])(y))
```

```
    except (ValueError, ZeroDivisionError) as e:
        print(type(e).__name__)
```

EXERCISE 2

Ledger

Implement `Account` with a class method `from_line("ada:100")`, a `transfer(to, amount)` method that raises `ValueError` when the balance is insufficient (checking before changing either account), and a class-level count of open accounts. Commands: `open name:balance`, `transfer from to amount` (prints `ok`, `insufficient funds` or `unknown account`), `report` (prints `name balance` in name order, then `accounts <count>`).

Input: commands. **Output:** one line per transfer, and the report lines.

```
open ada:100
open bob:20
transfer ada bob 30
transfer bob ada 500
transfer cy ada 1
report
```

prints

```
ok
insufficient funds
unknown account
ada 70
bob 50
accounts 2
```

INPUT 1

```
open ada:100
open bob:20
transfer ada bob 30
transfer bob ada 500
transfer cy ada 1
report
```

OUTPUT 1

```
ok
insufficient funds
unknown account
ada 70
bob 50
accounts 2
```

REFERENCE SOLUTION

```python
import sys


class Account:
    opened = 0

    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        type(self).opened += 1

    @classmethod
    def from_line(cls, text):
        name, balance = text.split(":")
        return cls(name, int(balance))

    def transfer(self, to, amount):
        if amount > self.balance:
            raise ValueError("insufficient funds")
        self.balance -= amount
        to.balance += amount


accounts = {}
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "open":
        acct = Account.from_line(args[0])
        accounts[acct.name] = acct
    elif cmd == "transfer":
        src, dst = accounts.get(args[0]), accounts.get(args[1])
        if src is None or dst is None:
            print("unknown account")
            continue
        try:
            src.transfer(dst, int(args[2]))
            print("ok")
        except ValueError:
            print("insufficient funds")
    elif cmd == "report":
        for name in sorted(accounts):
            print(name, accounts[name].balance)
        print(f"accounts {Account.opened}")
```

EXERCISE 3

Product catalogue

Define `@dataclass(frozen=True, order=True) Product` with fields `name: str`, `price_cents: int`, `qty: int` and a derived field `value: int = field(init=False, compare=False)` set in `__post_init__` with `object.__setattr__`. Read `n` products; print them ordered by value descending then name (an explicit key), then the first product with its quantity increased by one via `replace` (the value must be recomputed automatically), then `asdict` of that copy as JSON with sorted keys.

Input: `n`, then `n` lines `name price qty`. **Output:** `n` lines `name value`, then the replaced product's repr, then the JSON.

```
2
bolt 5 100
nut 2 250
```

prints

```
bolt 500
nut 500
Product(name='bolt', price_cents=5, qty=101, value=505)
{"name": "bolt", "price_cents": 5, "qty": 101, "value": 505}
```

INPUT 1

```
2
bolt 5 100
nut 2 250
```

OUTPUT 1

```
bolt 500
nut 500
Product(name='bolt', price_cents=5,
qty=101, value=505)
{"name": "bolt", "price_cents": 5,
"qty": 101, "value": 505}
```

REFERENCE SOLUTION

```python
import json
from dataclasses import dataclass, field, replace, asdict


@dataclass(frozen=True, order=True)
class Product:
    name: str
    price_cents: int
    qty: int
    value: int = field(init=False, compare=False)

    def __post_init__(self):
        object.__setattr__(self, "value", self.price_cents * self.qty)


n = int(input())
products = []
for _ in range(n):
    name, price, qty = input().split()
    products.append(Product(name, int(price), int(qty)))
for p in sorted(products, key=lambda p: (-p.value, p.name)):
    print(p.name, p.value)
bumped = replace(products[0], qty=products[0].qty + 1)
print(bumped)
print(json.dumps(asdict(bumped), sort_keys=True))
```

CHECKPOINT QUESTIONS

1. What does `__init__` return?

A. The new instance

B. `None` — it initialises an instance that `__new__` already created

C. The class

D. Whatever the last statement evaluates to

2. Which attribute assignment creates per-instance state?

A. `count = 0` in the class body

B. `self.count = 0` in `__init__`

C. `Account.count = 0`

D. `cls.count = 0` in a classmethod

3. What does `print([obj])` use to display `obj`?

A. `__str__`

B. `__repr__`

C. `__format__`

D. `__print__`

4. Which dunder makes `for x in obj` work best?

A. `__len__`

B. `__iter__` returning an iterator

C. `__next__`

D. `__contains__`

5. What is `Money.__rmul__` for?

A. `money * 3`

B. `3 * money`

C. `money *= 3`

D. `-money`

6. How is a read-only attribute made with a property?

A. `@property` with no setter

B. `@property(readonly=True)`

C. Naming it with two underscores

D. `__slots__`

7. Which decorator marks an alternative constructor?

A. `@staticmethod`

B. `@classmethod`

C. `@property`

D. `@constructor`

8. What does `@dataclass` do with a field written `items: list = field(default_factory=list)`?

A. Shares one list across instances

B. Calls `list()` for each new instance

C. Raises `ValueError`

D. Makes the field read-only

9. A dataclass without `frozen=True` but with the default `eq=True`: can its instances be set members?

A. Yes

B. No — `__hash__` is set to `None`

C. Only if they have no fields

D. Yes, hashed by identity

10. What does `type(self).count += 1` do inside `__init__`?

A. Creates an instance attribute

B. Increments the class attribute of the instance's actual class

C. Raises `AttributeError`

D. Increments a global

11. What should `__lt__` return for an incomparable `other`?

A. `False`

B. `NotImplemented`

C. `None`

D. `0`

12. Which is the sign of a class that should be a dataclass?

A. It has an invariant checked in every method

B. It stores fields and has an `__init__`, `__repr__` and `__eq__` that only copy and compare them

C. It manages a resource

D. It subclasses `dict`

ANSWER KEY · CHECKPOINT — CLASSES AND OBJECTS

1. B Returning a value from `__init__` is a `TypeError`; the instance is created before `__init__` runs.

2. B Assignment through `self` writes the instance's own dictionary; the others write the class.

3. B Containers show their elements' `repr`; only a direct `print(obj)` uses `__str__`.

4. B `__iter__` is the iteration protocol; `__getitem__` is only a fallback, and `__next__` belongs on the iterator, not the collection.

5. B The reflected operator runs when the left operand's `__mul__` returns `NotImplemented`.

6. A Without a setter, assignment raises `AttributeError`.

7. B It needs the class (`cls`) to build the instance so that subclasses get instances of themselves.

8. B The factory is invoked per instance, giving each its own list.

9. B Value equality on a mutable object cannot safely have a hash; freeze it or set `eq=False`.

10. B `type(self)` is the class, so the assignment targets the shared attribute — and for a subclass, that subclass's.

11. B Python then tries the reflected comparison and raises a clear `TypeError` if neither side can order the pair.

12. B Pure record classes are exactly what `@dataclass` generates; hand-written boilerplate adds bugs, not value.