Module 9 of 20

Inheritance, protocols and duck typing

Subclassing with super() and attribute lookup, duck typing with EAFP, abstract base classes and collections.abc, multiple inheritance with the MRO and mixins, typing.Protocol for structural typing, and composition over inheritance with Liskov, delegation and UserDict.

Python has three ways to say that objects share behaviour, and knowing which one a situation calls for is the skill this module teaches. Inheritance shares an implementation and declares an is-a relationship; duck typing shares nothing but a method name and is how most Python code actually composes; protocols and abstract base classes make the duck-typed contract explicit — to the type checker or at construction time. Behind all of them is one mechanism, attribute lookup along the method resolution order, and one design test, Liskov's: can the subclass go wherever the base goes without anyone noticing?

Inheritance basics covers `class Sub(Base)`, overriding and extending with `super()`, the lookup order, `isinstance` versus `type`, and `object`. Polymorphism and duck typing explains programming to behaviour, EAFP versus LBYL, `hasattr` / `getattr`, and the `isinstance` chain that a method per class replaces. Abstract base classes covers `abc.ABC`, `@abstractmethod`, template methods, and `collections.abc` with its mixin methods. Multiple inheritance covers C3 linearisation, the diamond, cooperative `__init__` with `**kwargs`, and mixins. Protocols covers `typing.Protocol`, `@runtime_checkable`, attribute protocols and the standard `Supports…` family. Composition over inheritance states Liskov with the `Square(Rectangle)` counter-example, delegation, why subclassing `dict` misbehaves and `UserDict`, and the strategy pattern.

The exercises are whole programs: an animal hierarchy with a shared repr, employees and managers with an extended `pay`, a chorus of unrelated ducks handled by EAFP, a recursive total over mixed data, shapes on an ABC with a template method, a sequence built from two methods, an MRO trace through a diamond, cooperative initialisation of mixins, a runtime-checkable `Closable`, a structural `Named` protocol satisfied by four unrelated shapes, a stack by composition versus a list subclass, and a case-insensitive `UserDict`. The checkpoint adds a shape catalogue on an ABC, a plugin registry validated by a protocol, and a lower-casing mapping with a composed stack.

CONTENTS

1	Inheritance basics — subclasses, super() and attribute lookup	14 min
2	Polymorphism and duck typing — EAFP, hasattr and programming to behaviour	13 min
3	Abstract base classes — abc and collections.abc	14 min
4	Multiple inheritance and the MRO — mixins and cooperative super()	14 min
5	Protocols and structural typing — typing.Protocol	13 min
6	Composition over inheritance — Liskov, delegation and wrapping built-ins	14 min
✓	Checkpoint — Inheritance, protocols and duck typing	25 min

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

LESSON 1 · 14 MIN

Inheritance basics — subclasses, super() and attribute lookup

Inheritance lets a class start from another: `class Dog(Animal)` gives `Dog` every attribute and method of `Animal` and lets it add or replace what differs. Python's version is simple in mechanism — attribute lookup walks from the instance to its class to the base classes — and the only real subtlety is initialisation: a subclass's `__init__` replaces the base's, so it must call `super().__init__()` explicitly or the base's attributes never exist. This lesson covers the syntax, what is inherited, overriding and extending, `super()`, `isinstance` / `issubclass`, the lookup order, and the `object` root that everything descends from.

The statement

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

    def speak(self):
        return "..."

    def describe(self):
        return f"{self.name} says {self.speak()}"

class Dog(Animal):
    def speak(self):                    # override
        return "woof"

class Puppy(Dog):
    def __init__(self, name, age_weeks):
        super().__init__(name)          # extend: run the base initialiser, th
        self.age_weeks = age_weeks

    def describe(self):                 # extend a method
        return super().describe() + f" (aged {self.age_weeks} weeks)"

p = Puppy("rex", 8)
p.describe()          # 'rex says woof (aged 8 weeks)'
```

`class Dog(Animal):` names the base in parentheses. `Dog` inherits `__init__`, `describe` and `name`; it overrides `speak`. `describe` in the base calls `self.speak()`, and `self` is a `Dog`, so the *override* runs — that is polymorphism, and it needs nothing declared: every method is virtual.

What is inherited

Everything in the class namespace: methods, class attributes, properties, dunders, class methods and static methods. Instance attributes are not "inherited" in the same sense — they are created by whichever `__init__` runs, which is why a subclass that defines its own `__init__` must call the base's, or `self.name` is never set and the first `describe()` fails with `AttributeError`.

---

super()

`super()` returns a proxy that looks up attributes starting *after the current class* in the method resolution order (next lesson but one). `super().__init__(name)` calls `Animal.__init__` with `self` bound automatically; `super().describe()` calls the base version of a method you are extending. Two rules. Call `super().__init__()` **first** in a subclass initialiser unless there is a reason not to, so the base's attributes exist before your code uses them. And pass the base what it expects — its parameters, not yours: `Puppy.__init__` takes `(name, age_weeks)` and forwards only `name`.

The old spelling `Animal.__init__(self, name)` works but hard-codes the base; `super()` follows the MRO and survives refactoring and multiple inheritance.

Attribute lookup

`p.describe` is resolved by searching, in order: `p.__dict__` (instance), `Puppy.__dict__`, `Dog.__dict__`, `Animal.__dict__`, `object.__dict__`. The first hit wins. That order — the class's `__mro__` — is what makes overriding work (the subclass is searched first) and what makes an instance attribute shadow a class attribute of the same name. `type(p)` is `Puppy`; `Puppy.__bases__` is `(Dog,)`; `Puppy.__mro__` is `(Puppy, Dog, Animal, object)`.

isinstance and issubclass

```
isinstance(p, Puppy)          # True
isinstance(p, Animal)         # True — an instance of a subclass is an instanc
isinstance(p, (Dog, int))     # True — any of a tuple
issubclass(Puppy, Animal)     # True
type(p) is Animal             # False — type() is exact; isinstance respects i
```

Use `isinstance`, not `type(x) == C`, when subclasses should count — which is almost always. `isinstance(True, int)` is `True` because `bool` subclasses `int`; `isinstance(3, float)` is `False`. Long `isinstance` chains that switch behaviour on the type are the thing polymorphism replaces (next lesson).

The object root

Every class inherits from `object`, explicitly or not: `class A:` and `class A(object):` are the same in Python 3. `object` supplies the default `__init__` (no arguments), `__repr__`, `__eq__` (identity), `__hash__`, `__str__` (falls back to `__repr__`) and the machinery of attribute access. That is why a class with no methods at all can be instantiated, printed and compared.

Overriding dunders and the repr

Overriding `__repr__` in a subclass usually means including the subclass's own fields; `type(self).__name__` makes a base-class repr correct for every subclass automatically:

```
class Animal:
    def __repr__(self):
        return f"{type(self).__name__}({self.name!r})"
```

`Puppy("rex", 8)` then prints as `Puppy('rex')` without `Puppy` defining anything. The same trick works in `__eq__` (`isinstance(other, type(self))`) and in class-method constructors (`cls(...)`).

Extending versus replacing

An override that does not call `super()` *replaces* the base behaviour; one that does *extends* it, before or after its own work. Both are legitimate, and the choice should be deliberate: `Cat.speak` replaces because the base answer is a placeholder, while `Puppy.describe` extends because the base text is still wanted. The trap is the accidental replacement — overriding a method that the base class relied on for bookkeeping (a `close()` that flushed, a `__setattr__` that validated) and dropping the call, so the base's invariant quietly stops being maintained.

Inheriting from built-ins

`class Stack(list)` is tempting and usually wrong: `list` has forty methods that bypass your rules (`insert`, slicing assignment, `extend`), and its C implementation does not call your overridden `__setitem__` from `append`. Prefer composition — hold a list in an attribute and expose the operations you mean — or subclass `collections.UserList` / `UserDict`, which are written in Python and route everything through the basic methods (lesson 6).

Pitfalls

- A subclass `__init__` that forgets `super().__init__()`, leaving base attributes unset.
- Calling `super().__init__()` with the subclass's extra arguments.
- `type(x) == C` where `isinstance` was meant.
- Overriding a method with a different signature, so code written for the base breaks on the subclass.
- Deep hierarchies for code reuse; three levels is a lot.
- Subclassing `dict` / `list` to add rules.

Key takeaways

- `class Sub(Base)` inherits the base's namespace; a subclass overrides by redefining and extends by calling `super()`.
- A subclass `__init__` must call `super().__init__(...)` with the base's arguments, first.
- Lookup walks instance → class → bases → `object`; the first match wins, so overrides and shadowing follow.
- `isinstance` respects inheritance; `type()` is exact; `bool` is an `int`.

Everything descends from `object`; use `type(self).__name__` in shared dunders; compose rather than subclass built-ins.

EXERCISE 1

Animals

Build `Animal(name)` with `speak()` returning `...`, `describe()` returning `<name> says <speak()>`, and a `__repr__` using `type(self).__name__` so every subclass prints correctly. `Dog` and `Cat` override `speak` (`woof`, `meow`); `Puppy(Dog)` takes an extra `age_weeks`, calls `super().__init__`, and extends `describe` with `(aged <n> weeks)`. Read lines `dog name`, `cat name`, `puppy name weeks` and print each animal's repr and description.

Input: lines. **Output:** two lines per animal: the repr, then the description.

```
dog rex
puppy bit 8
```

prints

```
Dog('rex')
rex says woof
Puppy('bit')
bit says woof (aged 8 weeks)
```

INPUT 1

```
dog rex
puppy bit 8
```

OUTPUT 1

```
Dog('rex')
rex says woof
Puppy('bit')
bit says woof (aged 8 weeks)
```

REFERENCE SOLUTION

```
import sys


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

    def speak(self):
        return "..."

    def describe(self):
        return f"{self.name} says {self.speak()}"

    def __repr__(self):
        return f"{type(self).__name__}({self.name!r})"


class Dog(Animal):
    def speak(self):
        return "woof"


class Cat(Animal):
    def speak(self):
        return "meow"


class Puppy(Dog):
    def __init__(self, name, age_weeks):
        super().__init__(name)
        self.age_weeks = age_weeks

    def describe(self):
        return super().describe() + f" (aged {self.age_weeks} weeks)"


for line in sys.stdin:
    kind, *args = line.split()
    if kind == "dog":
        animal = Dog(args[0])
    elif kind == "cat":
        animal = Cat(args[0])
    else:
        animal = Puppy(args[0], int(args[1]))
    print(repr(animal))
    print(animal.describe())
```

EXERCISE 2

Employees and managers

`Employee(name, salary)` has `pay()` returning the salary. `Manager(Employee)` adds a `bonus` and its `pay()` **extends** the base version with `super()`. Read staff lines `employee name salary` or `manager name salary bonus`; print each person's pay, then the total payroll, then how many objects are instances of `Employee` and how many of `Manager` (using `isinstance`).

Input: lines. **Output:** `<name> <pay>` per line, then `total <n>`, `employees <n>`, `managers <n>`.

```
employee ada 100
manager bob 200 50
```

prints

```
ada 100
bob 250
total 350
employees 2
managers 1
```

INPUT 1

```
employee ada 100
manager bob 200 50
```

OUTPUT 1

```
ada 100
bob 250
total 350
employees 2
managers 1
```

REFERENCE SOLUTION

```python
import sys


class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def pay(self):
        return self.salary


class Manager(Employee):
    def __init__(self, name, salary, bonus):
        super().__init__(name, salary)
        self.bonus = bonus

    def pay(self):
        return super().pay() + self.bonus


staff = []
for line in sys.stdin:
    kind, name, *nums = line.split()
    if kind == "manager":
        staff.append(Manager(name, int(nums[0]), int(nums[1])))
    else:
        staff.append(Employee(name, int(nums[0])))
for person in staff:
    print(person.name, person.pay())
print(f"total {sum(p.pay() for p in staff)}")
print(f"employees {sum(isinstance(p, Employee) for p in staff)}")
print(f"managers {sum(isinstance(p, Manager) for p in staff)}")
```

QUIZ

1. What happens?

```python
class A:
    def __init__(self):
        self.x = 1

class B(A):
    def __init__(self):
        self.y = 2

print(B().x)
```

A. `1`

B. `AttributeError` — `B.__init__` never called `A.__init__`

C. `2`

D. `None`

2. In the base class, `describe()` calls `self.speak()`. On a `Dog` instance, which `speak` runs?

A. `Animal.speak`

B. `Dog.speak` — lookup starts at the instance's actual class

C. Both

D. It depends on where `describe` is defined

3. What is `isinstance(True, int)`?

A. `False`

B. `True` — `bool` subclasses `int`

C. `TypeError`

D. `None`

4. What does `super()` inside `Puppy.__init__` refer to?

A. The `Animal` class

B. A proxy that starts lookup after `Puppy` in the MRO — here `Dog`, then `Animal`

C. The instance

D. The `object` class

5. Why is `class Stack(list)` usually a mistake?

A. Lists cannot be subclassed

B. It inherits every list method, so `insert(0, x)` and slicing can break the stack's rules; compose a list instead

C. It is slower

D. `super()` does not work with built-ins

ANSWER KEY · INHERITANCE BASICS — SUBCLASSES, SUPER() AND ATTRIBUTE LOOKUP

1. B A subclass initialiser replaces the base's; without `super().__init__()` the base attributes are never created.

2. B `self` is a `Dog`, and attribute lookup walks the MRO from `Dog`; every method is virtual.

3. B `isinstance` respects inheritance, and `bool` is a subclass of `int`. `type(True) is int` is `False`.

4. B `super()` follows the MRO of the instance's type rather than naming a parent, which is what makes it work with multiple inheritance.

5. B Inheritance exposes the whole base interface; composition exposes exactly the operations you write.

LESSON 2 · 13 MIN

Polymorphism and duck typing — EAFP, hasattr and programming to behaviour

"If it walks like a duck and quacks like a duck, it is a duck." Python does not ask an object what class it is before calling a method on it; it calls the method and either it works or it raises. That is *duck typing*, and it means polymorphism in Python needs no inheritance at all: any object with a `speak()` method can be passed to code that calls `speak()`. This lesson explains how that differs from class-based polymorphism, the EAFP style that goes with it (try the operation, handle the failure) versus LBYL (check first), the `hasattr` / `getattr` tools, the built-ins that are themselves duck-typed (`len`, `iter`, `sorted`), and where an explicit type check is still right.

Polymorphism without inheritance

```
class Dog:
    def speak(self):
        return "woof"

class Robot:                        # no common base class
    def speak(self):
        return "beep"

def chorus(things):
    return " ".join(t.speak() for t in things)

chorus([Dog(), Robot()])            # 'woof beep'
```

`chorus` is written against a *behaviour* — has a `speak()` — not a type. Java would need an interface both classes implement; Python needs only the method. The base class in the previous lesson gave `Dog` and `Puppy` a shared implementation to reuse, which is what inheritance is for; it was never needed for `chorus` to work.

The standard library is built this way. `len(x)` works on anything with `__len__`; `for x in y` on anything with `__iter__` or `__getitem__`; `with x:` on anything with `__enter__` / `__exit__`; `sorted(xs)` on anything whose elements have `__lt__`; `json.dump(obj, f)` on anything with a `write` method — a file, a socket wrapper, an

`io.StringIO`. A function that accepts "a file-like object" means exactly that: anything with the methods it calls.

EAFP versus LBYL

Look Before You Leap checks the precondition, then acts. **Easier to Ask Forgiveness than Permission** acts and handles the exception. Python idiom leans EAFP:

```
if key in d:                    # LBYL: two lookups, and a race if d can change
    value = d[key]
else:
    value = default

try:                            # EAFP: one lookup, atomic, and the failure path
    value = d[key]
except KeyError:
    value = default
```

EAFP is preferred when the failure is *exceptional* (the key is usually there), when the check would duplicate the operation, or when the check cannot be made reliably (a file may vanish between `os.path.exists` and `open`). LBYL is preferred when the failure is *common* — a `try` that raises on half the iterations is slower than an `if` — and when the check is cheap and clear (`if not xs: return`). For the specific case above, `d.get(key, default)` beats both. Module 10 covers exception handling in full; the design point here is that duck typing and EAFP go together: you do not check whether the duck can quack, you ask it to.

---

hasattr and getattr

When you genuinely need to know whether an object offers a behaviour before committing to a path:

```
if hasattr(obj, "close"):
    obj.close()

write = getattr(stream, "write", None)
if write is not None:
    write(text)

callable(obj)                    # has __call__
```

`getattr(obj, name, default)` returns the attribute or the default without raising; `hasattr` is `getattr` with a boolean result. These are the duck-typed versions of an `isinstance` check — they ask about capability, not lineage — and they are right for optional protocols: "flush if it can be flushed". A run of `hasattr` checks that dispatches to different code paths is still a type switch in disguise, and the same remedy applies: give each object a method and call it.

The isinstance chain, and its replacement

```
def area(shape):                          # the anti-pattern
    if isinstance(shape, Circle):
        return 3.14159 * shape.r ** 2
    elif isinstance(shape, Rect):
        return shape.w * shape.h
    ...

class Circle:                             # the replacement: each type knows i
    def area(self):
        return 3.14159 * self.r ** 2

def total_area(shapes):
    return sum(s.area() for s in shapes)
```

Adding a `Triangle` to the chain means editing `area`; adding it to the second design means writing one class. That is the Open/Closed idea in its Python form: behaviour lives with the data it belongs to, and the code that uses it never needs to know the list of types. When the set of types is closed and the operation does not belong to the types (formatting a value for output, say), `functools.singledispatch` (Module 11) or a `match` on class patterns (Module 3) is the honest way to write the switch.

When a type check is right

At a boundary, validating input: `if not isinstance(n, int): raise TypeError(...)`.

In a dunder deciding whether it can handle `other`: `if not isinstance(other, Money): return NotImplemented`.

Distinguishing text from a sequence of text: a `str` is iterable, so `for item in value` on a string iterates characters — `isinstance(value, str)` is the standard guard in functions that accept "one or many".

Dispatching on data that is not yours: JSON values, where `dict` / `list` /scalar have no methods to call.

Prefer `isinstance` against an abstract base or a protocol (next lessons) over a concrete class, so that anything with the behaviour qualifies.

Pitfalls

`isinstance` chains that switch behaviour on concrete types.

`type(x) == T`, which rejects subclasses.

Checking a method exists and then calling a *different* one.

EAFP with a bare `except:` that hides every error, not just the one expected.

Forgetting that a `str` quacks like a sequence of characters.

A `try` around a large block where only one call can raise the expected exception.

Key takeaways

Duck typing: code against behaviour (the methods you call), not lineage; no shared base class is required.

The built-ins are duck-typed through dunders — `len`, iteration, `with`, `sorted`, file-like objects.

EAFP (try, then handle) for exceptional failures and unreliable checks; LBYL for common, cheap conditions.

`hasattr` / `getattr(obj, name, default)` / `callable` ask about capability, not type.

Replace `isinstance` chains with a method per class; keep type checks for boundaries, dunders and the string-versus-sequence case.

EXERCISE 1

A chorus of ducks

`Dog`, `Robot` and `Duck` are unrelated classes that each have `speak()`; `Rock` has no `speak`. Read a line of class names, instantiate each, and print what each says using **EAFP**: call `speak()` inside `try` and print `(silent)` on `AttributeError`. No `isinstance` and no `hasattr`.

Input: one line of names from `dog robot duck rock`. **Output:** one word per name, space-separated.

```
dog rock robot
```

prints

```
woof (silent) beep
```

INPUT 1

```
dog rock robot
```

OUTPUT 1

```
woof (silent) beep
```

REFERENCE SOLUTION

```
class Dog:
    def speak(self):
        return "woof"


class Robot:
    def speak(self):
        return "beep"


class Duck:
    def speak(self):
        return "quack"


class Rock:
    pass


KINDS = {"dog": Dog, "robot": Robot, "duck": Duck, "rock": Rock}


def voice(thing):
    try:
        return thing.speak()
    except AttributeError:
        return "(silent)"


print(" ".join(voice(KINDS[name]()) for name in input().split()))
```

EXERCISE 2

Total over mixed data

Write `total(value)` that adds up a nested structure: an integer counts as itself, a string of digits is converted with `int`, and a list is summed recursively. Use EAFP: try `int(value)` first and, on `TypeError`, treat the value as an iterable. Read one literal (parse with `ast.literal_eval`) and print the total.

Input: one line holding a list literal of ints, digit strings and nested lists. **Output:** the total.

```
[1, "2", [3, "4"], 5]
```

prints

```
15
```

INPUT 1

```
[1, "2", [3, "4"], 5]
```

OUTPUT 1

```
15
```

REFERENCE SOLUTION

```python
import ast


def total(value):
    try:
        return int(value)
    except TypeError:
        return sum(total(v) for v in value)


print(total(ast.literal_eval(input())))
```

QUIZ

1. What does duck typing mean?

A. Every class must inherit from `Duck`

B. Code uses an object's methods without checking its type; anything with the right methods works

C. Types are checked at compile time

D. Only built-in types are allowed

2. Which is the EAFP spelling of a dictionary lookup with a fallback?

A. `if k in d: v = d[k] else: v = 0`

B. `try: v = d[k] except KeyError: v = 0`

C. `v = d[k] or 0`

D. `assert k in d`

3. When is LBYL preferable to EAFP?

A. Never

B. When the failure is common and the check is cheap and reliable

C. When the exception is a `KeyError`

D. Only in loops

4. What does `getattr(obj, "flush", None)` return when `obj` has no `flush`?

A. Raises `AttributeError`

B. `None`

C. `False`

D. An empty function

5. What replaces a long `isinstance` chain that picks behaviour by concrete type?

A. A longer chain

B. A method on each class, called polymorphically

C. `type()` comparisons

D. Global variables

ANSWER KEY · POLYMORPHISM AND DUCK TYPING — EAFP, HASATTR AND PROGRAMMING TO BEHAVIOUR

1. B Behaviour, not lineage, decides. `len(x)` works on anything with `__len__`.

2. B Attempt the operation and handle the specific failure; the first option is LBYL. (`d.get(k, 0)` is the shortest of all here.)

3. B Raising an exception on most iterations costs more than a test; when the check can be made cheaply and cannot race, checking first reads well.

4. B The third argument is the default returned instead of raising.

5. B Behaviour belongs with the data; adding a type then means adding a class, not editing every switch.

LESSON 3 · 14 MIN

Abstract base classes — abc and collections.abc

Duck typing answers "can this object do X?" at the moment X is attempted. An abstract base class answers it earlier: a class that declares `@abstractmethod`s cannot be instantiated until a subclass fills them in, so a missing method fails at construction rather than deep inside a call. The standard library's `collections.abc` goes further — implement the two or three core methods of `Sequence` or `Mapping` and the ABC supplies the rest. This lesson covers `abc.ABC` and `@abstractmethod`, abstract properties and class methods, `collections.abc` with its mixin methods, `register` for virtual subclasses, and `isinstance` against an ABC as the type check that respects duck typing.

Declaring an interface

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self) -> float: ...

    @abstractmethod
    def perimeter(self) -> float: ...

    def describe(self) -> str:                    # concrete: uses the abstr
        return f"{type(self).__name__}: area {self.area():.2f}, perimeter {sel

class Rect(Shape):
    def __init__(self, w, h):
        self.w, self.h = w, h
    def area(self):
        return self.w * self.h
    def perimeter(self):
        return 2 * (self.w + self.h)

Shape()          # TypeError: Can't instantiate abstract class Shape with abst
Rect(2, 3).describe()    # 'Rect: area 6.00, perimeter 10.00'
```

`ABC` is the base that switches the check on; `@abstractmethod` marks what subclasses must provide. A subclass that leaves one out is itself abstract and cannot be instantiated — the error names the missing methods, at the point of construction. Concrete methods on the ABC (`describe`) are the *template method* pattern: shared behaviour written once in terms of the abstract operations.

The `...` body is conventional for an abstract method; a docstring or `raise NotImplementedError` also works. An abstract method *may* have a body that subclasses call via `super()`.

Abstract properties and class methods

```
class Plugin(ABC):
    @property
    @abstractmethod
    def name(self) -> str: ...

    @classmethod
    @abstractmethod
    def from_config(cls, cfg): ...
```

Stack `@abstractmethod` innermost. A subclass may satisfy an abstract property with a plain class attribute `name = "csv"` or a property — either makes the name resolvable, which is all the check requires.

---

collections.abc

The ABCs in `collections.abc` describe the built-in protocols and, crucially, supply *mixin methods* derived from a few abstract ones:

ABC	You implement	You get for free
`Iterable`	`__iter__`	—
`Sized`	`__len__`	—
`Container`	`__contains__`	—
`Sequence`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, `count`
`MutableSequence`	+ `__setitem__`, `__delitem__`, `insert`	`append`, `extend`, `pop`, `remove`, `__iadd__`, …
`Mapping`	`__getitem__`, `__len__`, `__iter__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`
`MutableMapping`	+ `__setitem__`, `__delitem__`	`pop`, `popitem`, `clear`, `update`, `setdefault`
`Set`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__and__`, `__or__`, `isdisjoint`, …

```python
from collections.abc import Sequence

class Countdown(Sequence):
    def __init__(self, n):
        self.n = n
    def __len__(self):
        return self.n
    def __getitem__(self, i):
        if isinstance(i, slice):
            return [self[j] for j in range(*i.indices(self.n))]
        if i < 0:
            i += self.n
        if not 0 <= i < self.n:
            raise IndexError(i)
        return self.n - i

c = Countdown(3)
list(c), 2 in c, c.index(1), c.count(3), list(reversed(c))   # [3, 2, 1] True
```

Two methods, and the class is a full read-only sequence — with `index` and `count` you did not write, and `isinstance(c, Sequence)` true. `MutableMapping` is the way to build a dict-like

class with custom storage: implement five methods and `update`, `setdefault`, `pop` arrive.

---

isinstance against an ABC

The `collections.abc` classes recognise duck-typed objects through `__subclasshook__`: `isinstance(x, Iterable)` is true for anything with `__iter__`, whether or not it inherits from `Iterable`. That makes ABCs the right target for the boundary checks of the previous lesson:

```
from collections.abc import Iterable, Mapping

def flatten(items):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, (str, bytes)):
            yield from flatten(x)
        else:
            yield x

def merge(a: Mapping, b: Mapping) -> dict: ...
```

`Iterable` rather than `list` accepts tuples, sets, generators and custom classes; the `str` exclusion is the string-versus-sequence rule again. `Hashable`, `Callable`, `Sized` work the same way. For your own ABCs the hook is not automatic — a class must inherit, or be registered.

---

register: virtual subclasses

```
class Drawable(ABC):
    @abstractmethod
    def draw(self): ...

Drawable.register(SomeThirdPartyClass)      # promises it has draw(); nothing
isinstance(SomeThirdPartyClass(), Drawable) # True
```

Registration makes `isinstance` say yes for a class you cannot edit, without inheritance and without verification. It is rare, and the `Protocol` of lesson 5 is the modern answer to the same need.

ABC versus duck typing versus Protocol

An ABC is right when there is **shared implementation** to inherit (template methods, mixins from `collections.abc`) or when a **construction-time guarantee** matters (a plugin system that must fail fast on an incomplete plugin). Duck typing is right when the code simply calls methods and any provider will do. A `Protocol` (lesson 5) is right when you want the static checker to verify the duck typing without requiring inheritance. All three coexist: your `Shape` ABC's subclasses are also ducks to any code that calls `.area()`.

Pitfalls

- Forgetting `ABC` as a base (or `metaclass=ABCMeta`), so `@abstractmethod` is not enforced.
- Stacking `@abstractmethod` outermost over `@property` — it must be innermost.
- Subclassing `Sequence` but not handling `slice` in `__getitem__`.
- `isinstance(x, list)` at a boundary where `Sequence` or `Iterable` was meant.
- `isinstance(s, Iterable)` letting a string through as a sequence of characters.
- Expecting `register` to check anything.

Key takeaways

- `class X(ABC)` with `@abstractmethod`s cannot be instantiated until every abstract method is defined; the error names them at construction.
- Concrete methods on an ABC are template methods written against the abstract ones.
- `collections.abc.Sequence` / `Mapping` / `Set` supply the full protocol from two or three core methods.
- `isinstance(x, Iterable)` (and friends) recognise duck-typed objects; check against ABCs, not concrete types, and exclude `str` from sequence checks.
- Use an ABC for shared implementation or fail-fast guarantees; duck typing or a `Protocol` otherwise.

EXERCISE 1

Shapes on an ABC

Define `Shape(ABC)` with abstract `area()` and `perimeter()` and a concrete template method `describe()` returning `<ClassName>: area <a>, perimeter <p>` with two decimals. Implement `Rect(w, h)` and `Circle(r)`. Read shape lines and print their descriptions; a line `shape` attempts `Shape()` and prints `abstract` when `TypeError` is raised.

Input: lines `rect w h`, `circle r` or `shape`. **Output:** one line per input line.

```
rect 2 3
shape
circle 1
```

prints

```
Rect: area 6.00, perimeter 10.00
abstract
Circle: area 3.14, perimeter 6.28
```

INPUT 1

```
rect 2 3
shape
circle 1
```

OUTPUT 1

```
Rect: area 6.00, perimeter 10.00
abstract
Circle: area 3.14, perimeter 6.28
```

REFERENCE SOLUTION

```
import math
import sys
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def area(self):
        ...

    @abstractmethod
    def perimeter(self):
        ...

    def describe(self):
        return f"{type(self).__name__}: area {self.area():.2f}, perimeter {self.peri


class Rect(Shape):
    def __init__(self, w, h):
        self.w, self.h = w, h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)


class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return math.pi * self.r ** 2

    def perimeter(self):
        return 2 * math.pi * self.r


for line in sys.stdin:
    kind, *nums = line.split()
    if kind == "shape":
        try:
            Shape()
        except TypeError:
            print("abstract")
        continue
    values = [float(x) for x in nums]
```

```
    shape = Rect(*values) if kind == "rect" else Circle(*values)
    print(shape.describe())
```

EXERCISE 2

A sequence from two methods

Implement `Evens(n)` — the first `n` even numbers `0, 2, 4, …` — as a subclass of `collections.abc.Sequence`, defining only `__len__` and `__getitem__` (with negative indexes, `IndexError` past the ends, and slices). Then use the **inherited** mixin methods: read `n` and a query value `q`; print the elements, `q in`, `index(q)` (or `absent` on `ValueError`), `count(q)`, the reversed elements, and `isinstance(evens, Sequence)`.

Input: `n q`. **Output:** `elements <…>`, `in <bool>`, `index <i>` or `index absent`, `count <c>`, `reversed <…>`, `sequence <bool>`.

```
4 4
```

prints

```
elements 0 2 4 6
in True
index 2
count 1
reversed 6 4 2 0
sequence True
```

INPUT 1

```
4 4
```

OUTPUT 1

```
elements 0 2 4 6
in True
index 2
count 1
reversed 6 4 2 0
sequence True
```

REFERENCE SOLUTION

```
from collections.abc import Sequence


class Evens(Sequence):
    def __init__(self, n):
        self.n = n

    def __len__(self):
        return self.n

    def __getitem__(self, i):
        if isinstance(i, slice):
            return [self[j] for j in range(*i.indices(self.n))]
        if i < 0:
            i += self.n
        if not 0 ⩽ i < self.n:
            raise IndexError(i)
        return 2 * i


n, q = map(int, input().split())
e = Evens(n)
print("elements", " ".join(map(str, e)))
print(f"in {q in e}")
try:
    print(f"index {e.index(q)}")
except ValueError:
    print("index absent")
print(f"count {e.count(q)}")
print("reversed", " ".join(map(str, reversed(e))))
print(f"sequence {isinstance(e, Sequence)}")
```

QUIZ

1. What happens when you instantiate a subclass of an ABC that has not implemented one abstract method?

A. It works; the method raises when called

B. `TypeError` at instantiation naming the missing method

C. A warning

D. The base implementation is used

2. Which two methods make a `collections.abc.Sequence` subclass complete?

A. `__iter__` and `__next__`

B. `__getitem__` and `__len__`

C. `__contains__` and `__len__`

D. `index` and `count`

3. What is true of `isinstance(x, collections.abc.Iterable)`?

A. Only classes that inherit from `Iterable` pass

B. Any object with `__iter__` passes, via the ABC's subclass hook

C. Only lists and tuples pass

D. It raises for user classes

4. How is an abstract property declared?

A. `@abstractmethod` above `@property`

B. `@property` above `@abstractmethod`

C. `@abstractproperty` only

D. Properties cannot be abstract

5. Why exclude `str` in `isinstance(x, Iterable) and not isinstance(x, str)`?

A. Strings are not iterable

B. A string iterates its characters, which is rarely what a 'sequence of items' test means

C. `str` does not define `__iter__`

D. It is faster

ANSWER KEY · ABSTRACT BASE CLASSES — ABC AND COLLECTIONS.ABC

1. B The check happens at construction, which is the point: an incomplete implementation fails fast.

2. B The ABC derives iteration, containment, reversal, `index` and `count` from those two.

3. B The `collections.abc` classes recognise structural conformance for the built-in protocols.

4. B `@abstractmethod` must be the innermost decorator, applied to the function before `@property` wraps it.

5. B Text is iterable, so a flattening or one-or-many function would explode strings into characters without the guard.

LESSON 4 · 14 MIN

Multiple inheritance and the MRO — mixins and cooperative super()

A class may list several bases: `class Report(JsonMixin, CsvMixin, Base)`. Python resolves which method wins with the C3 *method resolution order*, a single linear order over all the bases that respects each class's own base order and puts subclasses before their parents. `super()` follows that order rather than "the parent", which is what makes cooperative initialisation across several bases possible — and what makes `super().__init__(**kwargs)` the correct spelling in a mixin. This lesson shows the MRO, the diamond, cooperative `super()`, the mixin pattern that is the one everyday use of multiple inheritance, and the cases to avoid.

The MRO

```
class A:
    def who(self):
        return "A"

class B(A):
    def who(self):
        return "B>" + super().who()

class C(A):
    def who(self):
        return "C>" + super().who()

class D(B, C):
    def who(self):
        return "D>" + super().who()

D.__mro__          # (D, B, C, A, object)
D().who()          # 'D>B>C>A'
```

`D(B, C)` inherits from both, and both inherit from `A` — the diamond. C3 linearisation gives `D, B, C, A, object`: `D` first, then its bases left to right, with `A` placed *after every class that inherits from it*, and `object` last. Every class appears once. `super()` in `B` therefore calls

`C.who`, not `A.who`, because `C` is next in *D*'s MRO. `super()` means "the next class in the MRO of the instance's type", and that is the whole story.

An inconsistent order — `class X(A, B)` where `B` is a subclass of `A` — raises `TypeError: Cannot create a consistent method resolution order`, because `A` cannot both precede and follow `B`.

Cooperative init

With several bases each needing initialisation, every class must call `super().__init__()` and forward what it does not consume:

```
class Base:
    def __init__(self, **kwargs):
        super().__init__(**kwargs)          # ends at object.__init__(), whicl

class Timestamped(Base):
    def __init__(self, *, created, **kwargs):
        super().__init__(**kwargs)
        self.created = created

class Named(Base):
    def __init__(self, *, name, **kwargs):
        super().__init__(**kwargs)
        self.name = name

class Event(Timestamped, Named):
    pass

e = Event(created="2024-05-01", name="deploy")
Event.__mro__       # (Event, Timestamped, Named, Base, object)
```

Each `__init__` takes its own arguments by keyword, passes the rest up, and sets its own attribute. The chain `Event → Timestamped → Named → Base → object` runs once through every class. Keyword-only parameters with `**kwargs` are what make this work regardless of the order the bases are combined in; positional arguments would have to agree on a position across unrelated classes.

Mixins

A mixin is a class that provides one capability, holds no state of its own (or very little), and is meant to be combined with a real base:

```
import json

class JsonMixin:
    def to_json(self):
        return json.dumps(self.__dict__, sort_keys=True)

class ReprMixin:
    def __repr__(self):
        fields = ", ".join(f"{k}={v!r}" for k, v in sorted(self.__dict__.items
        return f"{type(self).__name__}({fields})"

class Point(ReprMixin, JsonMixin):
    def __init__(self, x, y):
        self.x, self.y = x, y

Point(1, 2)              # Point(x=1, y=2)
Point(1, 2).to_json()    # '{"x": 1, "y": 2}'
```

Mixins go *before* the concrete base in the bases list so that their methods take precedence, are named `…Mixin` by convention, and depend on the host class only through a documented attribute or method (`self.__dict__` here). This is the everyday use of multiple inheritance in Python: `socketserver.ThreadingMixIn` , Django's view mixins, `unittest` test mixins — small, orthogonal, composable.

Reading an unfamiliar hierarchy

When a method behaves unexpectedly in a class with several bases, three lines settle it: `print(Cls.__mro__)` for the order, `Cls.method` (without calling) to see which class the attribute resolves to — its `__qualname__` names the defining class — and `inspect.getsource` when the body matters. Most surprises are a mixin listed after the base it was meant to override, or an `__init__` in the chain that does not call `super()` and so silently ends the cooperative sequence for every class after it.

Interfaces through ABCs

Combining several ABCs is multiple inheritance too: `class Deck(Sequence, Hashable)` promises both protocols. Since ABCs carry little or no state and their abstract methods are disjoint, they combine without cooperative-`__init__` concerns.

When to avoid it

Two concrete classes with real state and their own `__init__` signatures. Their initialisers will not compose without the `**kwargs` discipline, and their attributes may collide. Compose instead: hold one as an attribute of the other.

Inheriting for code reuse when there is no is-a relationship. A `Stack(list)` is not a list; a `Logger(Thread)` is not a thread. Composition again.

Deep or wide hierarchies that make `__mro__` a research task. If a reader must print the MRO to predict which method runs, the design has failed.

`ClassName.__mro__` and `ClassName.mro()` show the order; `inspect.getmro` too. Print it when in doubt — but design so that nobody has to.

Pitfalls

`Base.__init__(self)` by name in a mixin, breaking the chain for other bases.

Positional arguments in cooperative `__init__` s.

A mixin placed *after* the base, so the base's method wins.

Expecting `super()` to mean "my parent" in a diamond.

Two bases defining the same attribute name for different purposes.

Multiple inheritance where an attribute holding the other object would be clearer.

Key takeaways

The MRO is a C3 linearisation: subclass first, bases left to right, a shared base after all its subclasses; `Class.__mro__` shows it.

`super()` calls the next class in the *instance's* MRO, which in a diamond is a sibling, not the parent.

Cooperative `__init__` : every class calls `super().__init__(**kwargs)` and takes its own arguments by keyword.

Mixins add one capability, hold no state, go first in the bases list, and are the everyday use of multiple inheritance.

Combine ABCs freely; avoid combining stateful concrete classes — compose them instead.

EXERCISE 1

Trace the MRO

Define the diamond `A`, `B(A)`, `C(A)`, `D(B, C)`, each with `who()` returning its own letter followed by `>` and `super().who()` (`A` returns just `A`). Read class names, one per line; for each print the class's `__mro__` as the class names joined by spaces, then the result of `who()` on a fresh instance.

Input: lines, each `A`, `B`, `C` or `D`. **Output:** two lines per input line: `mro: …` and `who: …`.

```
D
B
```

prints

```
mro: D B C A object
who: D>B>C>A
mro: B A object
who: B>A
```

INPUT 1

```
D
B
```

OUTPUT 1

```
mro: D B C A object
who: D>B>C>A
mro: B A object
who: B>A
```

REFERENCE SOLUTION

```
import sys


class A:
    def who(self):
        return "A"


class B(A):
    def who(self):
        return "B>" + super().who()


class C(A):
    def who(self):
        return "C>" + super().who()


class D(B, C):
    def who(self):
        return "D>" + super().who()


CLASSES = {"A": A, "B": B, "C": C, "D": D}
for line in sys.stdin:
    cls = CLASSES[line.strip()]
    print("mro:", " ".join(k.__name__ for k in cls.__mro__))
    print("who:", cls().who())
```

EXERCISE 2

Cooperative initialisation

Write three mixins with **cooperative** initialisers — `Named` (keyword `name`), `Timestamped` (keyword `created`) and `Tagged` (keyword `tags`, a comma-separated string stored as a sorted list) — each taking its own argument by keyword, forwarding `**kwargs` with `super().__init__(**kwargs)`, and `Event(Tagged, Timestamped, Named)`. Read one line of `key=value` pairs, build `Event(**pairs)`, print its attributes in alphabetical order, then its MRO.

Input: one line with `name=…`, `created=…`, `tags=…` in any order. **Output:** `created …`, `name …`, `tags …` (tags space-separated), then `mro: …`.

```
name=deploy tags=b,a created=2024-05-01
```

prints

```
created 2024-05-01
name deploy
tags a b
mro: Event Tagged Timestamped Named object
```

INPUT 1

```
name=deploy tags=b,a created=2024-05-01
```

OUTPUT 1

```
created 2024-05-01
name deploy
tags a b
mro: Event Tagged Timestamped Named
object
```

REFERENCE SOLUTION

```python
class Named:
    def __init__(self, *, name, **kwargs):
        super().__init__(**kwargs)
        self.name = name


class Timestamped:
    def __init__(self, *, created, **kwargs):
        super().__init__(**kwargs)
        self.created = created


class Tagged:
    def __init__(self, *, tags, **kwargs):
        super().__init__(**kwargs)
        self.tags = sorted(tags.split(","))


class Event(Tagged, Timestamped, Named):
    pass


pairs = dict(tok.split("=", 1) for tok in input().split())
e = Event(**pairs)
print(f"created {e.created}")
print(f"name {e.name}")
print("tags", " ".join(e.tags))
print("mro:", " ".join(k.__name__ for k in Event.__mro__))
```

QUIZ

1. For `class D(B, C)` with `B(A)` and `C(A)`, what is `D.__mro__`?

A. `(D, B, A, C, object)`

B. `(D, B, C, A, object)`

C. `(D, A, B, C, object)`

D. `(D, C, B, A, object)`

2. In that diamond, what does `super().who()` inside `B.who` call when the instance is a `D`?

A. `A.who`

B. `C.who`

C. `D.who`

D. `object.who`

3. Why do cooperative `__init__`s take keyword-only parameters and `**kwargs`?

A. For speed

B. So each class can consume its own argument and forward the rest regardless of the order the mixins are combined in

C. Because positional arguments are not allowed in mixins

D. To avoid `super()`

4. Where should a mixin appear in a class's base list?

A. After the concrete base

B. Before the concrete base, so its methods take precedence

C. It does not matter

D. Only as the sole base

5. What does `class X(A, B)` raise when `B` is a subclass of `A`?

A. Nothing

B. `TypeError: Cannot create a consistent method resolution order`

C. `AttributeError`

D. `RecursionError`

ANSWER KEY · MULTIPLE INHERITANCE AND THE MRO — MIXINS AND COOPERATIVE SUPER()

1. B C3 puts the shared base `A` after every class that inherits from it, and bases left to right.

2. B `super()` proceeds to the next class in the *instance's* MRO, which after `B` is `C`.

3. B Positional arguments would need every unrelated class to agree on positions; keywords route themselves.

4. B The MRO searches bases left to right; a mixin meant to override must come first.

5. B `A` would have to precede `B` (base order) and follow it (subclass before base); C3 cannot satisfy both.

LESSON 5 · 13 MIN

Protocols and structural typing — typing.Protocol

Duck typing has always been how Python code works; `typing.Protocol` (3.8) is how a type checker can *verify* it. A protocol is a class that lists methods and attributes; any object that has them satisfies the protocol — no inheritance, no registration — and `mypy` will report a caller that passes something that does not. That is *structural* typing (shape decides) as opposed to *nominal* typing (declared lineage decides), and it is what type hints for duck-typed code should say. This lesson covers declaring a protocol, using it as a hint, `@runtime_checkable` for `isinstance`, protocols with attributes and generic protocols, the `Supports…` protocols in the standard library, and when to choose a protocol over an ABC.

Declaring and using a protocol

```
from typing import Protocol

class Speaker(Protocol):
    def speak(self) → str: ...

class Dog:                          # no inheritance from Speaker
    def speak(self) → str:
        return "woof"

class Robot:
    def speak(self) → str:
        return "beep"

def chorus(things: list[Speaker]) → str:
    return " ".join(t.speak() for t in things)

chorus([Dog(), Robot()])            # checker: fine — both have speak() → str
chorus([Dog(), 42])                 # checker: error — int has no speak
```

The protocol body lists the members with `...` bodies. Nothing changes at run time: `Dog` and `Robot` are ordinary classes, `chorus` calls `speak()` as before. The hint `list[Speaker]` tells the checker the shape, and the checker verifies every call site. Method signatures matter — a `speak(self, loud: bool)` would not match — and return types must be compatible.

runtime_checkable

By default `isinstance(x, Speaker)` raises `TypeError`, because a protocol is a static notion. `@runtime_checkable` allows it, checking that the *method names* exist (not their signatures):

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Closable(Protocol):
    def close(self) -> None: ...

if isinstance(resource, Closable):
    resource.close()
```

This is `hasattr` with a name and a hint, and it is the correct run-time counterpart of the static check. It only inspects presence: an object with a `close` *attribute* that is not callable would pass.

Attributes and properties in a protocol

```
class Named(Protocol):
    name: str                       # a data attribute

class HasArea(Protocol):
    @property
    def area(self) -> float: ...    # a read-only property (a plain attribute
```

An attribute in a protocol matches an instance attribute, a class attribute or a property of the same name and compatible type. This is how a hint says "anything with a `.name`" — the shape of a dataclass, a named tuple and a hand-written class all match.

The standard protocols

`typing` and `collections.abc` already define the ones most code needs, so write your own only for your own operations:

Protocol	Requires
`Iterable[T]`, `Iterator[T]`	`__iter__` / `__next__`
`Sized`, `Container[T]`, `Collection[T]`	`__len__`, `__contains__`, both plus `__iter__`
`Sequence[T]`, `Mapping[K, V]`	the sequence / mapping methods
`Callable[[A, B], R]`	`__call__` with that signature
`Hashable`	`__hash__`
`SupportsInt`, `SupportsFloat`, `SupportsIndex`, `SupportsAbs`	`__int__`, `__float__`, `__index__`, `__abs__`
`ContextManager[T]`	`__enter__` / `__exit__`

The `collections.abc` names are both ABCs (for inheritance and `isinstance`) and, to the checker, structural: a parameter hinted `Iterable[int]` accepts any class with `__iter__`, whether or not it inherits.

Generic protocols

```python
from typing import Protocol, TypeVar

T = TypeVar("T")

class Comparable(Protocol):
    def __lt__(self, other: "Comparable") -> bool: ...

class Stack(Protocol[T]):
    def push(self, item: T) -> None: ...
    def pop(self) -> T: ...

def smallest(xs: list[Comparable]) -> Comparable:
    return min(xs)
```

A protocol may take type parameters like any generic (Module 15 covers `TypeVar` in full). `Comparable` is the classic one — "anything with `__lt__`" — and is what `sorted`'s elements must be. (3.12's `class Stack[T](Protocol)` syntax is reading only on this track's runtime.)

Protocol versus ABC

	Protocol	ABC
Conformance	structural: having the members	nominal: inheriting (or `register`)
Enforced	by the type checker; `runtime_checkable` for `isinstance`	at instantiation (`TypeError` for missing methods)
Shared implementation	none — a protocol should be pure interface	yes: concrete and mixin methods
Third-party classes	conform automatically	must inherit or be registered
Use for	hints on parameters that accept "anything with these methods"	base classes you control that share code or must fail fast

A common combination: an ABC for your own hierarchy's shared behaviour, and a `Protocol` in the signature of the function that only needs one method — so the function also accepts objects from outside the hierarchy.

Pitfalls

- `isinstance(x, SomeProtocol)` without `@runtime_checkable` — `TypeError`.
- Expecting a runtime check to verify signatures; it checks names only.
- Putting implementation in a protocol; keep it an interface.
- Hinting a parameter with a concrete class when a protocol or ABC would accept more.
- Mismatched method signatures silently failing the structural match in the checker.
- Writing a protocol for something `collections.abc` already names.

Key takeaways

- A `Protocol` lists members; any object with them conforms — structural typing, verified by the type checker.
- `@runtime_checkable` allows `isinstance`, checking member names only.
- Protocols may declare attributes and properties, and may be generic.

Prefer the standard `Iterable`, `Sequence`, `Callable`, `Supports…` protocols to writing your own.

Protocol for hints and third-party conformance; ABC for shared implementation and construction-time enforcement.

EXERCISE 1

Closable

Define a `@runtime_checkable` protocol `Closable` with `close(self) → None`. Four unrelated classes are given: `File` and `Socket` have `close()`, `Buffer` has `flush()` only, `Const` has a non-callable attribute named `close`. Read class names; for each instance print whether `isinstance(obj, Closable)` holds and, if it does, try to call `close()` — printing `closed <Class>` on success or `not callable` if the call raises `TypeError`.

Input: one line of class names. **Output:** one line per name: `<Class> closable=<bool>` followed, when closable, by a space and `closed <Class>` or `not callable`.

```
File Buffer Const
```

prints

```
File closable=True closed File
Buffer closable=False
Const closable=True not callable
```

INPUT 1

```
File Buffer Const
```

OUTPUT 1

```
File closable=True closed File
Buffer closable=False
Const closable=True not callable
```

REFERENCE SOLUTION

```python
from typing import Protocol, runtime_checkable


@runtime_checkable
class Closable(Protocol):
    def close(self) -> None: ...


class File:
    def close(self):
        return None


class Socket:
    def close(self):
        return None


class Buffer:
    def flush(self):
        return None


class Const:
    close = 42


KINDS = {"File": File, "Socket": Socket, "Buffer": Buffer, "Const": Const}
for name in input().split():
    obj = KINDS[name]()
    ok = isinstance(obj, Closable)
    line = f"{name} closable={ok}"
    if ok:
        try:
            obj.close()
            line += f" closed {name}"
        except TypeError:
            line += " not callable"
    print(line)
```

EXERCISE 2

Anything with a name

Define a `@runtime_checkable` protocol `Named` with a data member `name: str`. Objects of four unrelated shapes are built from the input — a dataclass `User`, a `namedtuple` `Pet`, a plain class

`Thing` and a `dict` — and a function `greet(obj)` prints `hello <name>` when `isinstance(obj, Named)` and `nameless` otherwise. (A dict does not conform: a key is not an attribute.)

Input: lines `user <name>`, `pet <name>`, `thing <name>`, `dict <name>`. **Output:** one line per input line.

```
user ada
dict bob
pet rex
```

prints

```
hello ada
nameless
hello rex
```

INPUT 1

```
user ada
dict bob
pet rex
```

OUTPUT 1

```
hello ada
nameless
hello rex
```

REFERENCE SOLUTION

```
import sys
from collections import namedtuple
from dataclasses import dataclass
from typing import Protocol, runtime_checkable


@runtime_checkable
class Named(Protocol):
    name: str


@dataclass
class User:
    name: str


Pet = namedtuple("Pet", "name")


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


def greet(obj):
    if isinstance(obj, Named):
        print(f"hello {obj.name}")
    else:
        print("nameless")


for line in sys.stdin:
    kind, name = line.split()
    obj = {"user": User, "pet": Pet, "thing": Thing, "dict": lambda n: {"name": n}}[
    greet(obj)
```

QUIZ

1. How does a class come to satisfy a `typing.Protocol`?

A. By inheriting from it

B. By having the members the protocol lists, with compatible signatures — no inheritance needed

C. By calling `register`

D. By defining `__protocol__`

2. What does `isinstance(x, SomeProtocol)` do without `@runtime_checkable`?

A. Returns `True` for conforming objects

B. Raises `TypeError`

C. Returns `False`

D. Returns `None`

3. What does the runtime check verify?

A. Method signatures and return types

B. Only that attributes with the listed names exist

C. That the class inherits from the protocol

D. Docstrings

4. Which hint accepts any object with `__iter__` yielding ints?

A. `list[int]`

B. `Iterable[int]`

C. `tuple[int]`

D. `Sequence[int]`

5. When is an ABC the better choice than a protocol?

A. When third-party classes must conform without changes

B. When there is shared implementation to inherit or a construction-time guarantee is wanted

C. When only a type checker is used

D. Never

ANSWER KEY · PROTOCOLS AND STRUCTURAL TYPING — TYPING.PROTOCOL

1. B Protocols are structural; conformance is by shape, checked by the type checker.

2. B A plain protocol is a static concept; the decorator opts in to a run-time presence check.

3. B It is `hasattr` under the hood; an attribute of the right name but wrong kind still passes.

4. B `Iterable` is the structural protocol for iteration; `Sequence` additionally requires indexing and length.

5. B Protocols carry no implementation and enforce nothing at run time; ABCs do both.

LESSON 6 · 14 MIN

Composition over inheritance — Liskov, delegation and wrapping built-ins

Inheritance says *is-a*; composition says *has-a*. A `Car` has an `Engine`; it is not one. Most of the trouble people have with class hierarchies comes from using inheritance for *has-a* relationships because it is the quickest way to reuse methods — and the cost arrives later, when the subclass cannot honour the base class's contract. This lesson states the Liskov substitution principle with the standard broken example, shows delegation as the composition mechanism (explicit forwarding and `__getattr__`), explains why subclassing `dict` and `list` misbehaves and what `UserDict` / `UserList` fix, and closes with the strategy pattern as composition of behaviour.

Liskov: a subclass must be usable wherever its base is

```
class Rectangle:
    def __init__(self, w, h):
        self.w, self.h = w, h
    def set_width(self, w):
        self.w = w
    def area(self):
        return self.w * self.h

class Square(Rectangle):                # a square is-a rectangle, mathematicall
    def __init__(self, side):
        super().__init__(side, side)
    def set_width(self, w):             # …but must keep its sides equal
        self.w = self.h = w

def stretch(r: Rectangle):
    r.set_width(10)
    return r.area()                     # a caller of Rectangle expects w * h wi

stretch(Rectangle(2, 3))                # 30
stretch(Square(3))                      # 100 — the Square broke the caller's ex
```

`Square` inherits `set_width` and must override it to preserve its own invariant, and in doing so violates the *behavioural* contract of `Rectangle`: setting the width no longer leaves the height

alone. Code written against `Rectangle` is wrong for `Square`. That is the Liskov substitution principle failing: a subclass must not strengthen preconditions, weaken postconditions, or break invariants the base promised. The mathematical is-a is not a software is-a when the base is *mutable*. The fixes: make both immutable value types (then `Square` really can be a `Rectangle`), or make them siblings under a `Shape` interface with no shared mutation.

The test to apply before subclassing: *could every function that takes the base be handed the subclass without noticing?* If not, compose.

Delegation

Composition holds the other object and forwards the operations it chooses to expose:

```
class Stack:
    def __init__(self):
        self._items = []                    # has-a list

    def push(self, x):
        self._items.append(x)

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

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

The `Stack` exposes three operations; `insert(0, x)`, slicing and `sort` — which would break a stack — do not exist on it. Compare `class Stack(list)`: one line shorter, forty methods leakier. Composition's interface is exactly what you wrote.

When many methods must be forwarded unchanged, `__getattr__` delegates the rest (Module 16 covers the mechanism):

```
class LoggedList:
    def __init__(self, items):
        self._items = list(items)
        self.log = []

    def append(self, x):                    # the one method that differs
        self.log.append(f"append {x!r}")
        self._items.append(x)

    def __getattr__(self, name):            # everything else: forward to the li
        return getattr(self._items, name)
```

`__getattr__` is called only for attributes not found normally, so `append` is handled here and `sort`, `index`, `count` and the rest reach the list. One limit: dunders are *not* forwarded by `__getattr__`, because the interpreter looks them up on the type, so `len(logged)` still needs an explicit `__len__`. Forward the dunders you need by hand.

Why subclassing dict and list misbehaves

```
class UpperDict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key.upper(), value)

d = UpperDict()
d["a"] = 1              # {'A': 1}     — goes through __setitem__
d.update(b=2)           # {'A': 1, 'b': 2}  — update() is C code that bypasses
UpperDict(c=3)          # {'c': 3}     — so does the constructor
```

The built-in containers are implemented in C, and their methods call each other's C implementations, not your Python overrides. `dict.update`, `dict.__init__`, `setdefault`, `list.extend`, `list.__init__` all skip the overridden method. The standard library's answer is `collections.UserDict`, `UserList` and `UserString`: Python classes that hold a real container in `.data` and route *every* operation through the basic methods, so an override of `__setitem__` is honoured everywhere:

```python
from collections import UserDict

class UpperDict(UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key.upper(), value)

d = UpperDict(c=3)
d.update(b=2)
d                       # {'C': 3, 'B': 2}
```

For a mapping with custom storage, `collections.abc.MutableMapping` (lesson 3) is the other correct base. Subclass `dict` itself only to add methods that do not change how existing ones behave.

Strategy: composing behaviour

Inheritance fixes behaviour at class-definition time; composition can choose it at run time by holding a callable or an object:

```python
class Sorter:
    def __init__(self, key=None, reverse=False):
        self.key, self.reverse = key, reverse         # the strategy, as data

    def sort(self, xs):
        return sorted(xs, key=self.key, reverse=self.reverse)

class Pricer:
    def __init__(self, discount_rule):
        self.discount_rule = discount_rule            # a function or an obje

    def price(self, item):
        return self.discount_rule(item.base_price)
```

Where a subclass-per-variant design would produce `ByNameSorter`, `ByAgeSorter`, `ReverseByAgeSorter`, one class with a strategy field covers every combination, and the strategy is a plain function in most cases. Callbacks, key functions, comparators and handlers are all this pattern.

The decision, in one paragraph

Inherit when the subclass *is* a base in every context the base is used, shares a real implementation with it, and you control both. Compose when you want to reuse behaviour, when the relationship is has-a or uses-a, when the wrapped object's full interface should not leak, or when the behaviour should vary at run time. Default to composition; promote to inheritance when the Liskov test passes and the shared code is substantial.

Pitfalls

- Subclassing to reuse methods when the subclass cannot honour the base's contract.
- `class X(dict)` or `(list)` with overridden item methods — use `UserDict` / `UserList` or `MutableMapping`.
- Forgetting that `__getattr__` does not forward dunders.
- A hierarchy of subclasses that differ by one setting — make it a field.
- Exposing the composed object (`stack.items`) and losing the invariant.
- Deep hierarchies with `super()` chains nobody can predict.

Key takeaways

- Liskov: a subclass must be usable wherever the base is; a mutable `Square(Rectangle)` fails the test.
- Composition holds the other object and forwards only the operations it means to expose; `__getattr__` forwards the rest, but not dunders.
- Built-in containers' C methods bypass Python overrides; subclass `UserDict` / `UserList` or `MutableMapping` to change behaviour.
- Behaviour that varies is a strategy field — usually a function — not a subclass per variant.
- Default to composition; inherit for genuine is-a with substantial shared implementation.

EXERCISE 1

Stack by composition

Write `BadStack(list)` — a list subclass with `push` (append) and `peek` — and `Stack`, which **composes** a list and exposes only `push`, `pop`, `peek`, `__len__` and `__bool__`. Run the same

commands on both and print the results side by side; commands that only a list has (`insert 0 v`, `sort`) succeed on the bad stack and print `no such operation` for the composed one (use `getattr` and `AttributeError`). `pop` and `peek` on an empty stack print `empty`.

Input: commands `push v`, `pop`, `peek`, `insert i v`, `sort`, `show`. **Output:** for `pop`, `peek`, `show` and the failing operations, one line: `bad=<result> good=<result>` (`show` prints the elements bottom to top, comma-separated, or `-`).

```
push 3
push 1
insert 0 9
sort
show
pop
```

prints

```
bad=ok good=no such operation
bad=ok good=no such operation
bad=1,3,9 good=3,1
bad=9 good=1
```

INPUT 1

```
push 3
push 1
insert 0 9
sort
show
pop
```

OUTPUT 1

```
bad=ok good=no such operation
bad=ok good=no such operation
bad=1,3,9 good=3,1
bad=9 good=1
```

REFERENCE SOLUTION

```
import sys


class BadStack(list):
    def push(self, x):
        self.append(x)

    def peek(self):
        return self[-1]


class Stack:
    def __init__(self):
        self._items = []

    def push(self, x):
        self._items.append(x)

    def pop(self):
        return self._items.pop()

    def peek(self):
        return self._items[-1]

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

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

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


def run(stack, cmd, args):
    if cmd == "push":
        stack.push(int(args[0]))
        return "ok"
    if cmd in ("pop", "peek"):
        if not stack:
            return "empty"
        return str(getattr(stack, cmd)())
    if cmd == "show":
        return ",".join(map(str, stack)) if stack else "-"
    try:
        method = getattr(stack, cmd)
    except AttributeError:
        return "no such operation"
    method(*map(int, args))
```

```
    return "ok"


bad, good = BadStack(), Stack()
for line in sys.stdin:
    cmd, *args = line.split()
    b, g = run(bad, cmd, args), run(good, cmd, args)
    if cmd in ("pop", "peek", "show", "insert", "sort"):
        print(f"bad={b} good={g}")
```

EXERCISE 2

A case-insensitive mapping

Subclass `collections.UserDict` to make `CaseDict`, which folds keys with `casefold()` in `__setitem__`, `__getitem__`, `__delitem__` and `__contains__`. Because `UserDict` routes everything through those methods, `update`, the constructor and `get` all fold too. Commands: `set K v`, `get K` (`missing` on `KeyError`), `has K`, `update K=v K=v …`, `keys` (sorted).

Input: commands. **Output:** one line per `get`, `has` and `keys`.

```
set Name ada
get NAME
update AGE=3 name=bob
get name
keys
```

prints

```
ada
bob
age name
```

INPUT 1

```
set Name ada
get NAME
update AGE=3 name=bob
get name
keys
```

OUTPUT 1

```
ada
bob
age name
```

REFERENCE SOLUTION

```
import sys
from collections import UserDict


class CaseDict(UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key.casefold(), value)

    def __getitem__(self, key):
        return super().__getitem__(key.casefold())

    def __delitem__(self, key):
        super().__delitem__(key.casefold())

    def __contains__(self, key):
        return super().__contains__(key.casefold())


d = CaseDict()
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "set":
        d[args[0]] = args[1]
    elif cmd == "get":
        try:
            print(d[args[0]])
        except KeyError:
            print("missing")
    elif cmd == "has":
        print(args[0] in d)
    elif cmd == "update":
        d.update(dict(tok.split("=", 1) for tok in args))
    elif cmd == "keys":
        print(" ".join(sorted(d)))
```

QUIZ

1. Why does a mutable `Square(Rectangle)` violate Liskov substitution?

A. Squares have fewer sides

B. Code written for `Rectangle` may set the width and expect the height unchanged; a `Square` cannot honour that

C. `super()` cannot be used

D. Rectangles cannot be subclassed

2. What does `__getattr__` *not* forward when used for delegation?

A. Methods

B. Dunder methods such as `__len__`, which the interpreter looks up on the type

C. Attributes starting with `_`

D. Class attributes

3. Why does overriding `__setitem__` in a `dict` subclass not affect `update()`?

A. `update` is a class method

B. The C implementation of `dict` calls its own internals, bypassing Python overrides

C. `update` uses `__setattr__`

D. It does affect it

4. Which is the composition solution to 'a class whose sorting behaviour varies'?

A. A subclass per sort order

B. A `key` field holding a function

C. Multiple inheritance from each order

D. A global setting

5. What is the default recommendation?

A. Inherit whenever code can be reused

B. Compose by default; inherit for genuine is-a with substantial shared implementation

C. Never inherit

D. Always use mixins

ANSWER KEY · COMPOSITION OVER INHERITANCE — LISKOV, DELEGATION AND WRAPPING BUILT-INS

1. B The subclass must keep the base's behavioural promises; a square's invariant breaks one of them.

2. B Special-method lookup bypasses instance `__getattr__`; forward the dunders you need explicitly.

3. B `collections.UserDict` is written in Python and routes every operation through the basic methods, so overrides are honoured.

4. B Behaviour that varies at run time is a strategy — data, usually a callable — not a class hierarchy.

5. B Composition exposes exactly the interface you write and cannot break a base contract; inheritance is for real is-a relationships.

CHECKPOINT · 25 MIN

Checkpoint — Inheritance, protocols and duck typing

This checkpoint covers the whole module: subclassing with `super()` and attribute lookup, duck typing with EAFP and `hasattr`, abstract base classes and `collections.abc`, multiple inheritance with the MRO, cooperative `super()` and mixins, `typing.Protocol` with `runtime_checkable`, and composition over inheritance with Liskov, delegation and `UserDict`.

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 happens when a subclass defines `__init__` and does not call `super().__init__()`?
- Why does `describe()` in a base class run the subclass's `speak()`?
- What does `isinstance(x, Iterable)` check, and why exclude `str` from a sequence test?
- Which two methods make a class a full `collections.abc.Sequence`?
- For `class D(B, C)` where both inherit from `A`, what is `D.__mro__`, and what does `super()` in `B` call?
- What does `@runtime_checkable` change about a `Protocol`?
- Why does `dict.update` bypass an overridden `__setitem__`, and what class fixes it?

The three programs are a shape hierarchy on an ABC with a template method, a plugin registry that validates objects against a runtime-checkable protocol, and a case-folding mapping built on `UserDict` alongside a stack that composes a list.

EXERCISE 1

Shape catalogue

Define `Shape(ABC)` with abstract `area()`, a concrete `describe()` returning `<ClassName> <area with two decimals>`, and `__lt__` comparing areas so shapes sort. Implement `Rect(w, h)`,

`Circle(r)` and `Square(Rect)` (constructed with one side, passing it twice to `super().__init__`). Read shapes; print them sorted by area ascending (ties keep input order), then `total <sum of areas>` with two decimals, then `squares <count>` using `isinstance` against `Rect` **and** `Square`: `rects <n> squares <m>` where a square counts as both.

Input: lines `rect w h`, `circle r`, `square s`. **Output:** the sorted descriptions, then `total …`, then `rects … squares …`.

```
rect 2 3
square 2
circle 1
```

prints

```
Circle 3.14
Square 4.00
Rect 6.00
total 13.14
rects 2 squares 1
```

INPUT 1

```
rect 2 3
square 2
circle 1
```

OUTPUT 1

```
Circle 3.14
Square 4.00
Rect 6.00
total 13.14
rects 2 squares 1
```

REFERENCE SOLUTION

```
import math
import sys
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def area(self):
        ...

    def describe(self):
        return f"{type(self).__name__} {self.area():.2f}"

    def __lt__(self, other):
        return self.area() < other.area()


class Rect(Shape):
    def __init__(self, w, h):
        self.w, self.h = w, h

    def area(self):
        return self.w * self.h


class Circle(Shape):
    def __init__(self, r):
        self.r = r

    def area(self):
        return math.pi * self.r ** 2


class Square(Rect):
    def __init__(self, side):
        super().__init__(side, side)


shapes = []
for line in sys.stdin:
    kind, *nums = line.split()
    values = [float(x) for x in nums]
    shapes.append({"rect": Rect, "circle": Circle, "square": Square}[kind](*values))
for s in sorted(shapes):
    print(s.describe())
print(f"total {sum(s.area() for s in shapes):.2f}")
print(f"rects {sum(isinstance(s, Rect) for s in shapes)} squares {sum(isinstance(s,
```

EXERCISE 2

Plugin registry

A `@runtime_checkable` protocol `Plugin` requires a data member `name` and a method `run(x)`. Candidate classes are given: `Doubler` and `Squarer` conform, `Broken` lacks `run`, `Anonymous` lacks `name`. Read a line of class names to register — accept a candidate only if its instance satisfies the protocol, printing `registered <name attr>` or `rejected <ClassName>` — then read an integer and print `<name>: <result>` for every registered plugin in registration order.

Input: a line of class names, then an integer. **Output:** one line per candidate, then one line per registered plugin.

```
Doubler Broken Squarer Anonymous
5
```

prints

```
registered double
rejected Broken
registered square
rejected Anonymous
double: 10
square: 25
```

INPUT 1

```
Doubler Broken Squarer Anonymous
5
```

OUTPUT 1

```
registered double
rejected Broken
registered square
rejected Anonymous
double: 10
square: 25
```

REFERENCE SOLUTION

```
from typing import Protocol, runtime_checkable


@runtime_checkable
class Plugin(Protocol):
    name: str

    def run(self, x: int) -> int: ...


class Doubler:
    name = "double"

    def run(self, x):
        return 2 * x


class Squarer:
    name = "square"

    def run(self, x):
        return x * x


class Broken:
    name = "broken"


class Anonymous:
    def run(self, x):
        return x


CANDIDATES = {"Doubler": Doubler, "Squarer": Squarer, "Broken": Broken, "Anonymous":
registry = []
for cls_name in input().split():
    obj = CANDIDATES[cls_name]()
    if isinstance(obj, Plugin):
        registry.append(obj)
        print(f"registered {obj.name}")
    else:
        print(f"rejected {cls_name}")
x = int(input())
for plugin in registry:
    print(f"{plugin.name}: {plugin.run(x)}")
```

EXERCISE 3

Lower-casing mapping and a composed history

Build `LowerDict(UserDict)` whose keys are always stored lower-cased, and `History`, a class that composes a list to record every command as `<n>: <command line>` with `add(line)`, `last(k)` returning the last `k` entries (fewer if there are fewer) and `__len__`. Commands: `set K v`, `get K` (`missing` when absent), `del K` (`missing` when absent), `history k` (prints the last `k` entries, one per line) and `count` (prints the number of recorded commands). Every command, including `history` and `count`, is recorded before it is executed.

Input: commands. **Output:** one line per `get`, per failed `del`, per `count`, and `k` lines per `history`.

```
set Name ada
get NAME
del name
del name
history 2
count
```

prints

```
ada
missing
4: del name
5: history 2
6
```

INPUT 1

```
set Name ada
get NAME
del name
del name
history 2
count
```

OUTPUT 1

```
ada
missing
4: del name
5: history 2
6
```

REFERENCE SOLUTION

```
import sys
from collections import UserDict


class LowerDict(UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key.lower(), value)

    def __getitem__(self, key):
        return super().__getitem__(key.lower())

    def __delitem__(self, key):
        super().__delitem__(key.lower())

    def __contains__(self, key):
        return super().__contains__(key.lower())


class History:
    def __init__(self):
        self._entries = []

    def add(self, line):
        self._entries.append(f"{len(self._entries) + 1}: {line}")

    def last(self, k):
        return self._entries[-k:] if k > 0 else []

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


d = LowerDict()
history = History()
for line in sys.stdin:
    line = line.rstrip("\n")
    history.add(line)
    cmd, *args = line.split()
    if cmd == "set":
        d[args[0]] = args[1]
    elif cmd == "get":
        print(d[args[0]] if args[0] in d else "missing")
    elif cmd == "del":
        if args[0] in d:
            del d[args[0]]
        else:
            print("missing")
    elif cmd == "history":
        for entry in history.last(int(args[0])):
```

```
            print(entry)
    elif cmd == "count":
        print(len(history))
```

CHECKPOINT QUESTIONS

1. What does `super().__init__(name)` do in a subclass initialiser?

A. Creates a new base instance

B. Runs the next class's `__init__` in the MRO on the same instance, so its attributes get set

C. Copies the base's attributes

D. Nothing unless the base is abstract

2. What is `type(p) == Dog` when `p` is a `Puppy(Dog)`?

A. `True`

B. `False` — `type` is exact; `isinstance(p, Dog)` would be `True`

C. `TypeError`

D. `None`

3. Which style is EAFP?

A. `if hasattr(x, "close"): x.close()`

B. `try: x.close() except AttributeError: pass`

C. `if isinstance(x, Closable): x.close()`

D. `assert hasattr(x, "close")`

4. What happens when `Shape()` is called and `Shape` has an unimplemented `@abstractmethod`?

A. An instance with a missing method

B. `TypeError: Can't instantiate abstract class …`

C. `NotImplementedError` when the method is called

D. A warning

5. Which methods does `collections.abc.Mapping` require?

A. `__getitem__`, `__len__`, `__iter__`

B. `__getitem__` only

C. `keys` and `values`

D. `__setitem__` and `__delitem__`

6. In `class D(B, C)`, which class's method runs for a name defined in both `B` and `C`?

A. `C`'s

B. `B`'s — earlier in the MRO

C. Both

D. It raises an ambiguity error

7. What is a mixin?

A. A class with many fields

B. A small class providing one capability, meant to be combined with a real base and holding little or no state

C. An abstract class

D. A module

8. What does a `Protocol` with `name: str` require of a conforming object?

A. A method `name()`

B. An attribute, class attribute or property called `name`

C. Inheritance from the protocol

D. A `__name__`

9. Why is `@runtime_checkable` needed for `isinstance(x, MyProtocol)`?

A. It is not; all protocols support it

B. Protocols are static by default; the decorator enables a presence check of the member names

C. It makes signatures checked

D. It registers the class

10. Which base should a dict-like class use so that overriding `__setitem__` affects `update` and the constructor?

A. `dict`

B. `collections.UserDict` (or `collections.abc.MutableMapping`)

C. `object`

D. `list`

11. What does the Liskov substitution principle require?

A. Subclasses must add methods

B. A subclass must be usable wherever the base is, without breaking the base's promises

C. Subclasses must be immutable

D. Every class needs an ABC

12. Which delegation mechanism forwards *unknown* attribute reads to an inner object?

A. `__getattribute__`

B. `__getattr__`

C. `__dict__`

D. `super()`

ANSWER KEY · CHECKPOINT — INHERITANCE, PROTOCOLS AND DUCK TYPING

1. B There is one instance; each initialiser in the chain adds its attributes to it.

2. B `type()` returns the concrete class; inheritance is only respected by `isinstance` / `issubclass`.

3. B Ask forgiveness: attempt the call and handle the specific failure.

4. B ABCs enforce completeness at construction.

5. A From those three it derives `__contains__`, `keys`, `items`, `values`, `get` and `__eq__`.

6. B The MRO lists bases left to right; the first class with the attribute wins.

7. B Mixins compose behaviour; they are listed before the concrete base so their methods take precedence.

8. B Data members in a protocol match any attribute of that name.

9. B Without it `isinstance` raises `TypeError`; with it, names are checked, signatures are not.

10. B Both route every operation through the basic item methods; `dict`'s C code does not.

11. B Preconditions may not be strengthened, postconditions may not be weakened, invariants must hold.

12. B `__getattr__` is called only when normal lookup fails, which makes it the forwarding hook; `__getattribute__` intercepts everything.