Module 16 of 20

Decorators, descriptors and the data model

Decorators as functions wrapping functions with wraps, arguments and stacking; closures with cells, late binding and factories; the descriptor protocol behind properties and methods; the attribute hooks and reflection; classes as objects with type, __new__ and __init_subclass__; and the rest of the data model through a Vector.

Everything in this module rests on one fact stated in Module 1: functions and classes are objects. A decorator is a function applied to a function; a descriptor is an object that intercepts attribute access; a class is an instance of `type` that can be created from data and hooked when subclassed; an operator is a call to a method the class defines. These are the mechanisms behind `@property`, `@dataclass`, `@cache`, ORMs, plugin systems and every library that seems to extend the language — and once the mechanism is visible, using or building such a thing is ordinary code.

Decorators covers the desugaring, `@wraps`, the common wrappers, factories with arguments, stacking order and class decorators. Closures and late binding goes underneath to cells, then to factories, accumulators with `nonlocal`, callbacks and the point at which a closure becomes a class with `__call__`. Descriptors states the protocol and the lookup order, builds a validated attribute with `__set_name__`, and re-derives `property`, methods and `cached_property`. Attribute access covers `__getattr__`, `__getattribute__`, `__setattr__`, reflection and `__slots__`. Classes as objects covers `type(name, bases, ns)`, `__new__`, `__init_subclass__`, `__class_getitem__` and metaclasses in outline. The data model completes the dunder map and builds a `Vector`.

The exercises are whole programs: traced and counted functions, a `retry` factory, closure cells inspected, an accumulator pair, a `Positive` descriptor, a hand-written cached property, a dict-backed configuration with `__getattr__`, a frozen object with `__setattr__`, a plugin registry driven by `__init_subclass__` with classes created by `type()`, a singleton and an immutable subclass via `__new__`, a `Vector`, and a grid with tuple keys and `__missing__`. The checkpoint adds a decorator suite stacked both ways, a small validation framework, and the complete `Vector`.

CONTENTS

1	Decorators — functions that wrap functions	15 min
2	Closures and late binding — cells, factories and stateful callables	13 min
3	Descriptors — how properties, methods and validated attributes work	14 min
4	Attribute access — __getattr__, __getattribute__, __setattr__, __dict__ and __slots__	13 min
5	Classes as objects — type, __new__, __init_subclass__ and metaclasses in outline	14 min
6	The data model — the rest of the dunders, and a Vector that uses them	14 min
✓	Checkpoint — Decorators, descriptors and the data model	25 min

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

LESSON 1 · 15 MIN

Decorators — functions that wrap functions

A decorator is a function that takes a function and returns a function, and the `@name` line above a `def` is only syntax for `f = name(f)`. Everything else follows: a decorator can add behaviour before or after every call (logging, timing, retrying, caching), replace the function entirely, register it somewhere, or check its arguments — without touching the body. Module 4's closures and Module 11's `functools.wraps` were the preparation. This lesson builds decorators from the plain form up: the wrapper shape, `@wraps`, decorators with arguments (three levels of `def`), stacking order, class decorators, and the standard ones you already use.

The desugaring

```
def shout(fn):
    def wrapper(*args, **kwargs):
        result = fn(*args, **kwargs)
        return result.upper()
    return wrapper

@shout
def greet(name):
    return f"hello {name}"

# is exactly:
def greet(name):
    return f"hello {name}"
greet = shout(greet)

greet("ada")          # 'HELLO ADA'
```

`shout` receives the original function, defines `wrapper` — a closure over `fn` — and returns it; the name `greet` is rebound to `wrapper`. `*args, **kwargs` let the wrapper accept whatever the original does and pass it through. The decorator runs *once, at definition time*; the wrapper runs *on every call*.

@wraps

```python
from functools import wraps

def shout(fn):
    @wraps(fn)                      # copies __name__, __doc__, __qualname__,
    def wrapper(*args, **kwargs):
        return fn(*args, **kwargs).upper()
    return wrapper

greet.__name__                      # 'greet', not 'wrapper'
greet.__wrapped__                   # the original, for tests and introspecti
```

Without `@wraps`, every decorated function reports its name as `wrapper` in tracebacks, `help()` and test output, and `inspect.signature` shows `(*args, **kwargs)`. It is one line and it is not optional.

The common wrappers

```python
def logged(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f"calling {fn.__name__}{args}")
        result = fn(*args, **kwargs)
        print(f"{fn.__name__} returned {result!r}")
        return result
    return wrapper

def count_calls(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        wrapper.calls += 1          # state on the wrapper function object
        return fn(*args, **kwargs)
    wrapper.calls = 0
    return wrapper

def memoize(fn):
    cache = {}                      # state in the closure
    @wraps(fn)
    def wrapper(*args):
        if args not in cache:
            cache[args] = fn(*args)
        return cache[args]
    return wrapper
```

Before/after behaviour, a counter, a cache: the state lives either in the closure (`cache`) or as an attribute on the wrapper (`wrapper.calls`), and the wrapper always returns what the original returned unless transforming the result is the point.

Decorators with arguments

`@retry(3)` is a call that *returns* a decorator, so the function has three levels:

```python
def retry(times):                              # 1. takes the arguments
    def decorator(fn):                         # 2. takes the function
        @wraps(fn)
        def wrapper(*args, **kwargs):          # 3. runs on each call
            last = None
            for _ in range(times):
                try:
                    return fn(*args, **kwargs)
                except TransientError as e:
                    last = e
            raise last
        return wrapper
    return decorator

@retry(3)
def fetch(): ...
# fetch = retry(3)(fetch)
```

The outer function is a *decorator factory*. A decorator that works both bare (`@retry`) and with arguments (`@retry(3)`) is possible but confusing; pick one form.

Stacking

```python
@logged
@memoize
def slow(n): ...
# slow = logged(memoize(slow))
```

Decorators apply bottom-up: the one nearest the `def` wraps first, the top one wraps last and runs *outermost* on a call. Here `logged` sees every call, including the ones `memoize` answers from cache; swapped, only cache misses would be logged. Order matters whenever wrappers have side effects.

Decorating methods and classes

A decorator on a method receives the function before it is bound; `self` arrives as `args[0]` in the wrapper, so the generic `*args, **kwargs` shape works unchanged. A **class decorator**

takes the class and returns it (or a replacement), the shape of `@dataclass`, `@total_ordering` and `@runtime_checkable`:

```
def register(cls):
    REGISTRY[cls.__name__] = cls
    return cls

@register
class CsvExporter: ...

def add_repr(cls):
    def __repr__(self):
        fields = ", ".join(f"{k}={v!r}" for k, v in vars(self).items())
        return f"{type(self).__name__}({fields})"
    cls.__repr__ = __repr__
    return cls
```

Registering plugins, adding methods, validating a class's attributes — all by decorating the class at definition time.

A decorator as a class

Anything callable can decorate, so a class with `__init__(self, fn)` and `__call__(self, *args, **kwargs)` is a decorator whose state is ordinary attributes rather than closure cells:

```
import functools

class CountCalls:
    def __init__(self, fn):
        functools.update_wrapper(self, fn)     # the class-form of @wraps
        self.fn = fn
        self.calls = 0

    def __call__(self, *args, **kwargs):
        self.calls += 1
        return self.fn(*args, **kwargs)

@CountCalls
def ping(): ...
ping(); ping.calls          # 1
```

The class form reads better when the wrapper has several pieces of state or methods of its own (`reset`, `stats`); the function form is shorter for one behaviour. One caveat: a class-based decorator on a *method* is not a descriptor unless it defines `__get__`, so `self` is not bound — the function form (a closure) has that for free.

The ones you already use

`@property`, `@staticmethod`, `@classmethod` (descriptors, lesson 3); `@functools.cache`, `@lru_cache`, `@wraps`, `@singledispatch`, `@total_ordering`; `@dataclass`; `@contextmanager`; `@abstractmethod`; `@runtime_checkable`; `@unittest.mock.patch`; `pytest.fixture`; Flask's and FastAPI's route decorators. Every one is `name(thing)` returning a replacement — nothing more.

Pitfalls

- No `@wraps` — tracebacks and `help` show `wrapper`.
- A wrapper that forgets to `return` the result.
- A wrapper with a fixed signature (`def wrapper(x)`) that breaks on keyword arguments.
- Confusing the factory (`retry(3)`) with the decorator (`retry`).
- Stacking in the wrong order.
- Doing expensive work at decoration time by accident (it runs at import).

Key takeaways

- `@deco` above `def f` is `f = deco(f)`; the decorator runs once, the wrapper on every call.
- A wrapper takes `*args, **kwargs`, calls the original, returns its result, and carries `@wraps(fn)`.
- State lives in the closure or on the wrapper object; `@retry(3)` is a factory returning a decorator.
- Stacked decorators apply bottom-up and run outermost-first; class decorators take and return a class.
- Every standard decorator is the same mechanism.

EXERCISE 1

Traced and counted

Write two decorators with `functools.wraps`: `traced` prints `→ <name>(<args comma-separated>)` before the call and `← <name> = <result>` after; `counted` keeps `wrapper.calls` on the wrapper. Apply both to `add(a, b)` and `square(x)` as `@counted` over `@traced`. Run lines `add 2 3` / `square 4`, then print `calls add=<n> square=<n>` and `names <add.__name__> <square.__name__>`.

Input: call lines. **Output:** the trace lines, then the two summary lines.

```
add 2 3
square 4
```

prints

```
→ add(2, 3)
← add = 5
→ square(4)
← square = 16
calls add=1 square=1
names add square
```

INPUT 1

```
add 2 3
square 4
```

OUTPUT 1

```
→ add(2, 3)
← add = 5
→ square(4)
← square = 16
calls add=1 square=1
names add square
```

REFERENCE SOLUTION

```
import sys
from functools import wraps


def traced(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f"→ {fn.__name__}({', '.join(map(str, args))})")
        result = fn(*args, **kwargs)
        print(f"← {fn.__name__} = {result}")
        return result
    return wrapper


def counted(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        return fn(*args, **kwargs)
    wrapper.calls = 0
    return wrapper


@counted
@traced
def add(a, b):
    return a + b


@counted
@traced
def square(x):
    return x * x


for line in sys.stdin:
    name, *nums = line.split()
    {"add": add, "square": square}[name](*map(int, nums))
print(f"calls add={add.calls} square={square.calls}")
print(f"names {add.__name__} {square.__name__}")
```

EXERCISE 2

A retry factory

Write `retry(times)` — a decorator factory — whose wrapper calls the function up to `times` times, printing `attempt <n> failed: <message>` for each `TransientError`, returning the first successful

result, and re-raising the last error when all attempts fail. Decorate `fetch`, which fails the first `k` times (read from input) then returns `"data"`. Read `k times` and print `result <value>` or `gave up`, then `fetch.__name__`.

Input: `k times`. **Output:** the attempt lines, the outcome, then the name.

```
2 3
```

prints

```
attempt 1 failed: flaky
attempt 2 failed: flaky
result data
name fetch
```

INPUT 1

```
2 3
```

OUTPUT 1

```
attempt 1 failed: flaky
attempt 2 failed: flaky
result data
name fetch
```

REFERENCE SOLUTION

```
from functools import wraps


class TransientError(Exception):
    pass


def retry(times):
    def decorator(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            last = None
            for attempt in range(1, times + 1):
                try:
                    return fn(*args, **kwargs)
                except TransientError as e:
                    last = e
                    print(f"attempt {attempt} failed: {e}")
            raise last
        return wrapper
    return decorator


k, times = map(int, input().split())
failures = [k]


@retry(times)
def fetch():
    if failures[0] > 0:
        failures[0] -= 1
        raise TransientError("flaky")
    return "data"


try:
    print(f"result {fetch()}")
except TransientError:
    print("gave up")
print(f"name {fetch.__name__}")
```

QUIZ

1. What does this expand to?

```
@deco
def f(): ...
```

A. `deco(f())`

B. `f = deco(f)`

C. `f = deco`

D. `deco.f = f`

2. Without `@wraps`, what is `f.__name__` after decoration?

A. `'f'`

B. `'wrapper'`

C. `None`

D. `'deco'`

3. How many nested functions does `@retry(3)` need?

A. One

B. Two

C. Three: the factory, the decorator, the wrapper

D. Four

4. For `@a` above `@b` above `def f`, which runs outermost on a call?

A. `b`

B. `a`

C. Neither

D. They alternate

5. What does a class decorator receive and return?

A. An instance

B. The class, returning the class or a replacement

C. The metaclass

D. A dict

ANSWER KEY · DECORATORS — FUNCTIONS THAT WRAP FUNCTIONS

1. B The decorator is applied once, at definition time, and the name is rebound to its result.

2. B The name is rebound to the wrapper function; `wraps` copies the metadata across.

3. C The factory takes the arguments and returns a decorator; the decorator takes the function and returns the wrapper.

4. B `f = a(b(f))`: `b` wraps first, `a` wraps the result and sees every call first.

5. B `@dataclass` and `@total_ordering` are class decorators of exactly this shape.

LESSON 2 · 13 MIN

Closures and late binding — cells, factories and stateful callables

A closure is a function that carries the variables of the scope it was created in, and Module 4 introduced it along with the trap: every closure created in a loop sees the loop variable's *final* value. This lesson goes underneath — what a cell is, how `__closure__` shows it, why late binding is the only consistent rule — and then covers the design patterns that closures make possible: factories that configure behaviour, accumulators with `nonlocal`, callbacks that remember context, and the point at which a closure with state should become a class with `__call__`.

Cells

```
def make_counter():
    count = 0
    def step():
        nonlocal count
        count += 1
        return count
    return step

c = make_counter()
c.__closure__                       # (<cell at 0x…: int object at 0x…>,)
c.__closure__[0].cell_contents      # 0
c(); c()
c.__closure__[0].cell_contents      # 2
c.__code__.co_freevars              # ('count',)
```

When the compiler sees that `step` refers to `count` from an enclosing function, it allocates `count` in a *cell* — a small box — instead of an ordinary local slot, and both `make_counter` and `step` read and write the same cell. `step.__closure__` is the tuple of those cells; `co_freevars` names them. The function object keeps the cells alive after `make_counter` returns, which is why the counter works: the frame is gone, the cell is not.

Late binding, precisely

A closure holds the *cell*, not the value that was in it at creation. Reading the variable happens when the closure runs:

```
fs = []
for i in range(3):
    fs.append(lambda: i)
[f() for f in fs]                    # [2, 2, 2] — one cell for i, read after

def make(i):
    return lambda: i                 # each call creates a NEW cell for its i
fs = [make(i) for i in range(3)]
[f() for f in fs]                    # [0, 1, 2]
```

The loop version has one `i` variable — one cell — shared by all three lambdas. The factory version calls a function per iteration, and each call has its own local `i`, so each lambda gets its own cell. The default-argument fix (`lambda i=i: i`) works differently: it copies the *value* into the lambda's own parameter at creation time. Both are correct; the factory is the one to prefer when the closure is more than a line, because it names what is being captured.

Factories

A factory configures a function and returns it — the closure holds the configuration:

```
def make_validator(lo, hi):
    def valid(x):
        return lo ⩽ x ⩽ hi
    return valid

def make_formatter(prefix, width):
    def fmt(value):
        return f"{prefix}{value:>{width}}"
    return fmt

percent = make_validator(0, 100)
money = make_formatter("$", 8)
```

Compared with a class with `__init__` and one method, the factory is shorter and the result is a plain function usable anywhere a function is expected (`sorted(key=…)`, `map`, callbacks). `functools.partial` (Module 11) is the factory for the case where the configuration is just fixed arguments.

Accumulators and nonlocal

```
def make_stats():
    total, count = 0.0, 0
    def add(x):
        nonlocal total, count
        total += x
        count += 1
    def mean():
        return total / count if count else 0.0
    return add, mean

add, mean = make_stats()
```

Two closures over the same cells make a tiny object with two methods and private state. `nonlocal` is needed for the rebinding (`total += x`); reading alone would not need it, and mutating a shared list (`values.append(x)`) would not either, because `append` does not rebind.

Callbacks that remember

```
def on_click(label):
    def handler(event):
        print(f"{label} clicked at {event.x},{event.y}")
    return handler

for name in ("ok", "cancel"):
    button(name).bind(on_click(name))     # each handler remembers its own lab
```

GUI toolkits, async frameworks and event systems hand you `event` and nothing else; the closure is how the handler knows *which* button, *which* request, *which* row. The loop above is the factory pattern again — `on_click(name)` per iteration — and the late-binding bug would be `bind(lambda e: print(name))`.

When a closure should be a class

A closure with one piece of state and one behaviour is ideal. When it grows — several pieces of state, several operations, a need to inspect or reset the state, a `__repr__` for debugging — a class with `__call__` keeps the callable interface and gains the rest:

```
class Counter:
    def __init__(self, start=0):
        self.count = start
    def __call__(self):
        self.count += 1
        return self.count
    def reset(self):
        self.count = 0

c = Counter()
c(); c()                                # callable like the closure
c.count, c.reset()                      # inspectable and resettable, unlike the
```

Both are *stateful callables*; the class is the honest form once the state has a name worth reading.

Pitfalls

- Closures created in a loop capturing the loop variable.
- Forgetting `nonlocal` and getting `UnboundLocalError` on `+=`.
- Closures over a mutable object that another part of the program also changes.
- A closure used as a cheap object that should have been a class.
- Relying on `__closure__` in production code (it is for introspection and debugging).
- Expecting a closure to copy a value; it captures the variable.

Key takeaways

- Enclosed variables live in cells shared by the outer function and the closure; `__closure__` and `co_freevars` show them.
- A closure reads its cells when called — late binding; a factory call per iteration or a default argument fixes loop capture.
- Factories configure functions; two closures over shared cells make a small object; `nonlocal` is for rebinding.

Callbacks use closures to remember context the framework does not pass.

When state and operations multiply, a class with `__call__` replaces the closure.

EXERCISE 1

Inside the cell

Write `make_counter(start)` returning a closure `step()` that increments and returns a `count` with `nonlocal`. Read `start` and `n`; print `step.__code__.co_freevars`, the cell's contents via `step.__closure__[0].cell_contents` before any call, the results of `n` calls, and the cell's contents afterwards. Then show late binding: build a list of three lambdas in a loop over `i` and a list of three via a factory, and print what each list returns.

Input: `start n`. **Output:** `freevars <tuple>`, `cell <v>`, `calls <values>`, `cell <v>`, `loop <values>`, `factory <values>`.

```
10 3
```

prints

```
freevars ('count',)
cell 10
calls 11 12 13
cell 13
loop 2 2 2
factory 0 1 2
```

INPUT 1

```
10 3
```

OUTPUT 1

```
freevars ('count',)
cell 10
calls 11 12 13
cell 13
loop 2 2 2
factory 0 1 2
```

REFERENCE SOLUTION

```
def make_counter(start):
    count = start

    def step():
        nonlocal count
        count += 1
        return count
    return step


start, n = map(int, input().split())
step = make_counter(start)
print(f"freevars {step.__code__.co_freevars}")
print(f"cell {step.__closure__[0].cell_contents}")
print("calls", " ".join(str(step()) for _ in range(n)))
print(f"cell {step.__closure__[0].cell_contents}")
loop = []
for i in range(3):
    loop.append(lambda: i)
factory = [(lambda v: (lambda: v))(i) for i in range(3)]
print("loop", " ".join(str(f()) for f in loop))
print("factory", " ".join(str(f()) for f in factory))
```

EXERCISE 2

An accumulator pair

Write `make_stats()` returning two closures over shared cells: `add(x)` (uses `nonlocal` to update a total and a count) and `summary()` returning `count=<n> mean=<m>` with the mean to two decimals (`0.00` when empty). Process `add x` and `summary` lines. Then create a **second** pair and add one value to it to show the state is separate: print its summary last.

Input: lines. **Output:** one line per `summary`, then `second count=1 mean=<x>`.

```
add 2
add 4
summary
add 9
summary
```

prints

```
count=2 mean=3.00
count=3 mean=5.00
second count=1 mean=100.00
```

INPUT 1

```
add 2
add 4
summary
add 9
summary
```

OUTPUT 1

```
count=2 mean=3.00
count=3 mean=5.00
second count=1 mean=100.00
```

REFERENCE SOLUTION

```
import sys


def make_stats():
    total, count = 0.0, 0

    def add(x):
        nonlocal total, count
        total += x
        count += 1

    def summary():
        mean = total / count if count else 0.0
        return f"count={count} mean={mean:.2f}"

    return add, summary


add, summary = make_stats()
for line in sys.stdin:
    cmd, *rest = line.split()
    if cmd == "add":
        add(float(rest[0]))
    else:
        print(summary())
add2, summary2 = make_stats()
add2(100)
print("second", summary2())
```

QUIZ

1. What does a closure store for an enclosed variable?

A. A copy of the value at creation

B. A cell shared with the enclosing scope, read when the closure runs

C. A string name

D. Nothing

2. Why does a factory call per iteration fix loop capture?

A. It copies the loop variable

B. Each call creates its own local — a new cell — for the closure to capture

C. It disables late binding

D. It does not

3. When is `nonlocal` required?

A. To read an enclosing variable

B. To rebind an enclosing variable (`+=` , `=`) from the inner function

C. To call the outer function

D. Always in closures

4. What does `f.__closure__` contain?

A. The source code

B. A tuple of cells for the enclosed variables (or `None`)

C. The outer function

D. The default arguments

5. When should a stateful closure become a class?

A. Never

B. When state and operations multiply and need inspection, reset or a `repr`

C. When it is called more than once

D. When it uses `nonlocal`

ANSWER KEY · CLOSURES AND LATE BINDING — CELLS, FACTORIES AND STATEFUL CALLABLES

1. B That is late binding: the variable's current value is read at call time.

2. B The loop has one variable; the factory has one per call.

3. B Reading needs nothing; mutating a shared list needs nothing; rebinding needs the declaration.

4. B `f.__code__.co_freevars` gives their names in the same order.

5. B A class with `__call__` keeps the callable interface and gains the rest.

LESSON 3 · 14 MIN

Descriptors — how properties, methods and validated attributes work

`@property` is not magic, and neither is the way `obj.method` becomes a bound method. Both are *descriptors*: objects with `__get__` (and optionally `__set__` / `__delete__`) that the attribute machinery consults when they are found on a class. Understanding the protocol explains `property`, `classmethod`, `staticmethod`, `functools.cached_property`, dataclass fields, ORM columns and every "attribute that computes or validates" you will meet. This lesson states the protocol, walks through the lookup order it plugs into, builds a validated-attribute descriptor with `__set_name__`, re-derives `property` from it, and explains data versus non-data descriptors — the distinction that decides who wins between the instance dict and the class.

The protocol

```
class Descriptor:
    def __get__(self, obj, objtype=None): ...     # obj is the instance, or Nor
    def __set__(self, obj, value): ...            # optional: makes it a *data*
    def __delete__(self, obj): ...                # optional
    def __set_name__(self, owner, name): ...      # optional: told the attribut
```

A descriptor is any object defining `__get__`; it takes effect only when it is a *class* attribute and is looked up through an instance or the class. `obj.attr` with a descriptor on the class calls `type(obj).__dict__["attr"].__get__(obj, type(obj))` instead of returning the descriptor itself.

The lookup order

`obj.x` proceeds:

Look for `x` on `type(obj)` and its MRO. If it is a **data descriptor** (has `__set__` or `__delete__`), call its `__get__` — it wins over everything.

Otherwise, if `x` is in `obj.__dict__`, return that.

Otherwise, if the class attribute is a **non-data descriptor** (only `__get__`), call its `__get__`.

Otherwise return the class attribute as is; failing that, `__getattr__` (next lesson); failing that, `AttributeError`.

Assignment `obj.x = v` checks the class for a data descriptor first (calling its `__set__`), else writes `obj.__dict__["x"]`. This ordering is why a `property` (data descriptor) cannot be shadowed by an instance attribute, while a method (non-data) can.

A validated attribute

```
class Positive:
    def __set_name__(self, owner, name):
        self.name = name                        # "price"
        self.storage = "_" + name               # "_price"

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self                         # class access returns the desc
        return getattr(obj, self.storage)

    def __set__(self, obj, value):
        if value <= 0:
            raise ValueError(f"{self.name} must be positive, got {value}")
        setattr(obj, self.storage, value)

class Item:
    price = Positive()
    qty = Positive()

    def __init__(self, price, qty):
        self.price = price                      # goes through Positive.__set__
        self.qty = qty

Item(5, 0)          # ValueError: qty must be positive, got 0
Item.price          # <Positive object> — obj is None
```

One descriptor class, reused for every attribute that needs the rule, with `__set_name__` telling each instance which attribute it guards. The per-object value is stored on the object under a private name, because the descriptor itself is shared by every instance of the class — storing the value on the descriptor would make every `Item` share one price.

property is a descriptor

```
class property:                                    # simplified
    def __init__(self, fget=None, fset=None, fdel=None):
        self.fget, self.fset, self.fdel = fget, fset, fdel
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)
    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)
    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel)
```

`@property` stores the getter in a data descriptor; `@x.setter` returns a new property with the setter added. That is the whole mechanism, and it is why a property with no setter refuses assignment (its `__set__` raises) and why it cannot be overridden by writing to `__dict__`.

Methods are descriptors too

A plain function has `__get__`: `function.__get__(obj, cls)` returns a *bound method* with `obj` fixed as the first argument. That is the step that turns `Class.method` (a function) into `instance.method` (a bound method) — a non-data descriptor, which is why an instance attribute of the same name can shadow a method. `staticmethod` wraps a function in a descriptor whose `__get__` returns the function unchanged; `classmethod`'s `__get__` binds the class instead of the instance. All three are the same protocol with different `__get__`s.

cached_property

```
class cached_property:                          # simplified
    def __init__(self, func):
        self.func = func
    def __set_name__(self, owner, name):
        self.name = name
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        value = self.func(obj)
        obj.__dict__[self.name] = value         # store on the instance…
        return value
```

A *non-data* descriptor that writes the computed value into the instance dict: on the next access, step 2 of the lookup finds the instance attribute first and the descriptor never runs again. Deleting the instance attribute clears the cache. The absence of `__set__` is the design.

Where descriptors appear

Dataclass fields with defaults, ORM columns (`Column(Integer)` in SQLAlchemy, `models.CharField` in Django), `enum` members, `unittest.mock` attributes, validation libraries, lazily loaded resources. Whenever an attribute on a class does something at access time, a descriptor is doing it.

Pitfalls

- Storing per-instance state on the descriptor object, which is shared.
- Forgetting `if obj is None: return self`, so class-level access crashes.
- Expecting an instance attribute to override a property (it cannot — data descriptors win).
- Defining a descriptor as an *instance* attribute (the protocol only fires on the class).
- Omitting `__set__` when validation is wanted (a non-data descriptor is bypassed by assignment).
- Reinventing `property`, `cached_property` or `functools.total_ordering`.

Key takeaways

A descriptor is a class attribute with `__get__` (and optionally `__set__` / `__delete__`); attribute access on an instance calls those methods.

Data descriptors (with `__set__`) beat the instance dict; non-data descriptors lose to it — the order behind properties, methods and `cached_property`.

`__set_name__` tells a descriptor its attribute name; store per-instance values on the instance, not on the descriptor.

`property`, `staticmethod`, `classmethod` and plain functions are all descriptors with different `__get__`s.

Reach for a descriptor when the same access-time rule applies to several attributes or classes.

EXERCISE 1

A validated attribute

Write a descriptor `Positive` with `__set_name__`, `__get__` (returning the descriptor itself when accessed on the class) and `__set__` (raising `ValueError("<name> must be positive, got <v>")` for values `⩽ 0`), storing each value on the instance under `_<name>`. `Item` declares `price = Positive()` and `qty = Positive()`. Process `set <attr> <value>` and `get <attr>` lines on one item created as `Item(1, 1)`; print errors' messages and values; finally print `type(Item.price).__name__`.

Input: lines. **Output:** one line per `get` and per rejected `set`, then the class-access line.

```
set price 5
set qty 0
get price
get qty
```

prints

```
qty must be positive, got 0
5
1
Positive
```

INPUT 1

```
set price 5
set qty 0
get price
get qty
```

OUTPUT 1

```
qty must be positive, got 0
5
1
Positive
```

REFERENCE SOLUTION

```
import sys


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

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

    def __set__(self, obj, value):
        if value <= 0:
            raise ValueError(f"{self.name} must be positive, got {value}")
        setattr(obj, self.storage, value)


class Item:
    price = Positive()
    qty = Positive()

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


item = Item(1, 1)
for line in sys.stdin:
    cmd, attr, *rest = line.split()
    if cmd == "set":
        try:
            setattr(item, attr, int(rest[0]))
        except ValueError as e:
            print(e)
    elif cmd == "get":
        print(getattr(item, attr))
print(type(Item.price).__name__)
```

EXERCISE 2

A cached property by hand

Implement `Cached`, a non-data descriptor (only `__get__` and `__set_name__`) that calls the wrapped function once per instance, stores the result in the instance's `__dict__` under the attribute name, and returns it — so later reads never reach the descriptor. Apply it to `Report.total`, whose body prints `computing` and sums the rows. Read rows, then commands `total` (print it) and `invalidate` (delete the instance attribute so the next read recomputes).

Input: a line of integers, then commands. **Output:** `computing` lines as they happen and the totals.

```
1 2 3
total
total
invalidate
total
```

prints

```
computing
6
6
computing
6
```

INPUT 1

```
1 2 3
total
total
invalidate
total
```

OUTPUT 1

```
computing
6
6
computing
6
```

REFERENCE SOLUTION

```
import sys


class Cached:
    def __init__(self, func):
        self.func = func

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        value = self.func(obj)
        obj.__dict__[self.name] = value
        return value


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

    @Cached
    def total(self):
        print("computing")
        return sum(self.rows)


report = Report(list(map(int, input().split())))
for line in sys.stdin:
    cmd = line.strip()
    if cmd == "total":
        print(report.total)
    elif cmd == "invalidate":
        report.__dict__.pop("total", None)
```

QUIZ

1. What makes an object a descriptor?

A. Being a class attribute

B. Defining `__get__` (and optionally `__set__` / `__delete__`) and being found on the class during lookup

C. Having a docstring

D. Being a property

2. Which wins: an instance attribute or a data descriptor of the same name?

A. The instance attribute

B. The data descriptor

C. Whichever was set last

D. It raises

3. Why does `cached_property` deliberately lack `__set__`?

A. To save memory

B. So that the value it stores in the instance dict shadows it on later reads

C. Because properties cannot be set

D. An oversight

4. What does `__set_name__` receive?

A. The instance and the value

B. The owner class and the attribute name, at class creation

C. Nothing

D. The metaclass

5. What turns `instance.method` into a bound method?

A. `__call__`

B. The function's own `__get__`, which binds the instance as the first argument

C. `__init__`

D. The metaclass

ANSWER KEY · DESCRIPTORS — HOW PROPERTIES, METHODS AND VALIDATED ATTRIBUTES WORK

1. B The protocol fires only for class attributes; an instance attribute with `__get__` is just an attribute.

2. B Data descriptors are checked first; that is why a property cannot be shadowed by writing to `__dict__`.

3. B A non-data descriptor loses to the instance dict — which is exactly the caching mechanism.

4. B It lets one descriptor class serve many attributes, each knowing its name.

5. B Plain functions are non-data descriptors; `staticmethod` and `classmethod` are descriptors with different `__get__`s.

LESSON 4 · 13 MIN

Attribute access — __getattr__, __getattribute__, __setattr__, __dict__ and __slots__

Every dot in Python is a call: `obj.x` runs `type(obj).__getattribute__(obj, "x")`, and `obj.x = v` runs `__setattr__`. Three hooks let a class intercept those calls — `__getattr__` for names that were not found, `__getattribute__` for every read, `__setattr__` / `__delattr__` for every write — and with `__dict__`, `getattr` / `setattr` / `hasattr` and `__slots__` they are the toolkit for proxies, dynamic attributes, immutable objects and attribute-based APIs. This lesson gives each hook its exact trigger, the recursion trap in the write hooks, the reflection functions, and the judgement of when interception is worth its opacity.

---

getattr: only on failure

```
class Config:
    def __init__(self, data):
        self._data = data

    def __getattr__(self, name):                # called ONLY when normal loo
        try:
            return self._data[name]
        except KeyError:
            raise AttributeError(name) from None

cfg = Config({"host": "db", "port": 5432})
cfg.host, cfg.port                              # 'db', 5432 — dict keys as a
cfg._data                                       # a real attribute: __getattr
cfg.missing                                     # AttributeError — as it shou
```

`__getattr__` is the fallback: normal lookup (instance dict, class, descriptors, bases) runs first, and only a failure reaches it. That makes it cheap and safe — real attributes are unaffected — and the right tool for delegation (Module 9's `__getattr__` forwarding to a wrapped object) and for exposing dict keys or remote fields as attributes. Two rules: raise `AttributeError` for names you do not handle, so `hasattr` and `getattr(obj, name, default)` keep working;

and remember that dunder lookups (`len(obj)`, `obj[i]`) bypass it, because the interpreter looks special methods up on the *type*.

---

getattribute: on every read

```
class Traced:
    def __getattribute__(self, name):
        value = super().__getattribute__(name)   # NEVER self.name here — inf
        print(f"read {name}")
        return value
```

`__getattribute__` intercepts *every* attribute read, including those inside its own body — `self.x` inside it calls itself again, forever. Always delegate to `super().__getattribute__` (or `object.__getattribute__(self, name)`). It is rarely the right tool: it is slower than every other approach and hides ordinary reads; `__getattr__`, a descriptor or a property almost always does the job.

---

setattr and delattr: on every write

```
class Frozen:
    def __init__(self, **fields):
        for k, v in fields.items():
            object.__setattr__(self, k, v)       # bypass our own hook during
        object.__setattr__(self, "_frozen", True)

    def __setattr__(self, name, value):
        if getattr(self, "_frozen", False):
            raise AttributeError(f"{type(self).__name__} is immutable")
        object.__setattr__(self, name, value)

    def __delattr__(self, name):
        raise AttributeError("immutable")
```

`__setattr__` runs on every assignment, including `self.x = …` in `__init__`, so the hook must write through `object.__setattr__` (or `super().__setattr__`) to avoid calling itself. This is how frozen dataclasses refuse assignment (`FrozenInstanceError` is an `AttributeError`), how validation on every field is done without a descriptor per field, and how a proxy forwards writes. The same recursion rule applies to `__delattr__`.

dict and vars

```
p = Point(1, 2)
p.__dict__                          # {'x': 1, 'y': 2}
vars(p) is p.__dict__               # True
p.__dict__["z"] = 3                 # adds an attribute, bypassing __setattr__
Point.__dict__                      # a read-only mappingproxy of the class nar
```

The instance dict is the ordinary storage; `vars(obj)` is the polite spelling. Writing to it directly skips the hooks, which is occasionally the point (a frozen class's constructor) and usually a hack.

The reflection functions

```
getattr(obj, "x")                   # obj.x; AttributeError if absent
getattr(obj, "x", default)          # with a fallback
setattr(obj, "x", 5)                # obj.x = 5 — through __setattr__
delattr(obj, "x")
hasattr(obj, "x")                   # True/False; internally getattr + except A
dir(obj)                            # the names, including inherited; __dir__ c
```

These take the attribute name as a *string*, which is what configuration-driven code needs: `setattr(self, key, value)` for each key of a dict, `getattr(module, function_name)` for a plugin, `getattr(self, f"handle_{command}")` for dispatch by name. A `hasattr` check followed by `getattr` is two lookups; `getattr` with a default is one.

slots again

`__slots__ = ("x", "y")` replaces the per-instance `__dict__` with fixed slots — smaller, faster to access, and a typo in an attribute name is an `AttributeError` rather than a silent new attribute (Module 8). The hooks above still apply (slots are descriptors underneath). Subclasses without their own `__slots__` regain a `__dict__`; `__slots__ = ()` in a subclass keeps the restriction. With `__slots__`, `vars(obj)` raises, so code that reflects over instances must use `dir` or the class's `__slots__`.

Dynamic attributes: when and when not

Exposing dict keys, environment variables or remote fields as attributes (`cfg.host`) reads well and is what `types.SimpleNamespace` does for free: `SimpleNamespace(**d)` gives a plain object with those attributes and a readable `repr`. The cost is opacity: an editor cannot complete `cfg.host`, a type checker cannot verify it, a typo becomes a run-time `AttributeError` (or worse, a silently created attribute on a proxy without `__setattr__`). Use dynamic attributes at boundaries where the field set is genuinely open; use a dataclass where it is known.

Pitfalls

- `self.x` inside `__getattribute__` or `self.x = v` inside `__setattr__` — infinite recursion.
- `__getattr__` that returns `None` for unknown names instead of raising `AttributeError`, breaking `hasattr`.
- Expecting `__getattr__` to intercept `len()`, `+` or indexing.
- Writing to `__dict__` to dodge validation.
- `__slots__` on a class whose subclasses forget it, silently reinstating `__dict__`.
- Dynamic attributes for data whose shape is fixed and known.

Key takeaways

- `__getattr__` runs only when normal lookup fails — the delegation and dict-as-attributes hook; raise `AttributeError` for the rest.
- `__getattribute__` runs on every read and must delegate to `super()`; it is rarely the tool.
- `__setattr__` / `__delattr__` run on every write, including in `__init__`; write through `object.__setattr__`.
- `getattr` / `setattr` / `hasattr` / `delattr` take names as strings; `vars` / `__dict__` is the storage; `SimpleNamespace` is the ready-made attribute bag.
- Special methods bypass instance hooks; `__slots__` removes `__dict__` and catches typos.

EXERCISE 1

Keys as attributes

Write `Config` holding a dict in `_data` with a `__getattr__` that serves the dict's keys as attributes and raises `AttributeError("no setting <name>")` otherwise. Build it from the first line's `key=value` pairs. Then process `get <name>` (print the value or the error message), `has <name>` (`hasattr`), and `real` (print `type(cfg._data).__name__` to show a real attribute never reaches the hook).

Input: the pairs line, then commands. **Output:** one line per command.

```
host=db port=5432
get host
get user
has port
has user
real
```

prints

```
db
no setting user
True
False
dict
```

INPUT 1

```
host=db port=5432
get host
get user
has port
has user
real
```

OUTPUT 1

```
db
no setting user
True
False
dict
```

REFERENCE SOLUTION

```python
import sys


class Config:
    def __init__(self, data):
        self._data = data

    def __getattr__(self, name):
        try:
            return self._data[name]
        except KeyError:
            raise AttributeError(f"no setting {name}") from None


cfg = Config(dict(tok.split("=", 1) for tok in input().split()))
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "get":
        try:
            print(getattr(cfg, args[0]))
        except AttributeError as e:
            print(e)
    elif cmd == "has":
        print(hasattr(cfg, args[0]))
    elif cmd == "real":
        print(type(cfg._data).__name__)
```

EXERCISE 2

Frozen after construction

Write `Frozen(**fields)` whose `__init__` sets the fields through `object.__setattr__` and then marks the instance frozen; afterwards `__setattr__` and `__delattr__` raise `AttributeError("immutable")`. Build one from the first line's `key=value` pairs, then process `set k v`, `del k` (print `immutable` on failure) and `get k` (`getattr` with default `missing`). Finally print the instance's `vars` keys in order.

Input: the pairs line, then commands. **Output:** the command results, then `fields <keys>`.

```
x=1 y=2
set x 5
get x
del y
get y
```

prints

```
immutable
1
immutable
2
fields x y _frozen
```

INPUT 1

```
x=1 y=2
set x 5
get x
del y
get y
```

OUTPUT 1

```
immutable
1
immutable
2
fields x y _frozen
```

REFERENCE SOLUTION

```
import sys


class Frozen:
    def __init__(self, **fields):
        for key, value in fields.items():
            object.__setattr__(self, key, value)
        object.__setattr__(self, "_frozen", True)

    def __setattr__(self, name, value):
        if getattr(self, "_frozen", False):
            raise AttributeError("immutable")
        object.__setattr__(self, name, value)

    def __delattr__(self, name):
        raise AttributeError("immutable")


obj = Frozen(**dict(tok.split("=", 1) for tok in input().split()))
for line in sys.stdin:
    cmd, *args = line.split()
    try:
        if cmd == "set":
            setattr(obj, args[0], args[1])
        elif cmd == "del":
            delattr(obj, args[0])
        elif cmd == "get":
            print(getattr(obj, args[0], "missing"))
    except AttributeError as e:
        print(e)
print("fields", " ".join(vars(obj)))
```

QUIZ

1. When is `__getattr__` called?

A. On every attribute read

B. Only when normal lookup fails to find the name

C. Only for dunder names

D. On writes

2. What is wrong with `def __setattr__(self, n, v): self.__dict__[n] = v` … `self.x = 1` inside `__getattribute__`?

A. Nothing

B. `self.x` inside `__getattribute__` calls `__getattribute__` again — infinite recursion

C. It is too slow

D. It only affects `__dict__`

3. Why must `__getattr__` raise `AttributeError` for unknown names?

A. Style

B. So that `hasattr` returns `False` and `getattr(obj, n, default)` returns the default

C. To log the failure

D. It must return `None` instead

4. Does `len(obj)` reach `__getattr__` when `__len__` is not defined?

A. Yes

B. No — special methods are looked up on the type, bypassing instance hooks

C. Only for old-style classes

D. Only with `__slots__`

5. What does `types.SimpleNamespace(**d)` give?

A. A dict

B. A plain object with the dict's keys as attributes and a readable repr

C. A dataclass

D. A frozen object

ANSWER KEY · ATTRIBUTE ACCESS — __GETATTR__, __GETATTRIBUTE__, __SETATTR__, __DICT__ AND __SLOTS__

1. B `__getattribute__` is the every-read hook; `__getattr__` is the fallback.

2. B Every read hook must delegate to `super().__getattribute__`; every write hook to `object.__setattr__`.

3. B Those helpers work by catching `AttributeError`.

4. B Forward the dunders you need explicitly when delegating.

5. B It is the ready-made attribute bag for the dict-as-attributes need.

LESSON 5 · 14 MIN

Classes as objects — type, __new__, __init_subclass__ and metaclasses in outline

A class is an object: `type(Dog)` is `type`, `Dog` has attributes, can be passed to functions, created at run time and modified after creation. That fact is the basis of every framework that registers, validates or generates classes — ORMs, serialisers, plugin systems, `dataclass` itself. This lesson covers `type` as the class factory (`type(name, bases, namespace)`), `__new__` versus `__init__` and the singleton it enables, `__init_subclass__` as the modern hook for "do something when a subclass is defined", `__class_getitem__` behind `list[int]`, and metaclasses — what they are, what they are for, and why `__init_subclass__` or a class decorator is almost always the better tool.

type is the class of classes

```
class Dog:
    sound = "woof"

type(Dog)                          # <class 'type'>
type(type)                         # <class 'type'> — it is its own class
Dog.__name__, Dog.__bases__        # 'Dog', (<class 'object'>,)
Dog.__dict__["sound"]              # 'woof' — the namespace
Dog.legs = 4                       # classes are mutable
isinstance(Dog, type)              # True
```

`type` is the *metaclass* of every ordinary class: the thing that, called with a name, a tuple of bases and a namespace dict, produces a class. The `class` statement is a call to it:

```
Cat = type("Cat", (object,), {"sound": "meow", "speak": lambda self: self.sour
Cat().speak()                      # 'meow'
```

Building classes from data — a table's columns, a schema, a config — is this call. `types.new_class` is the fuller API; `type(...)` is enough for most cases.

new and init

Instantiation is two steps: `Dog.__new__(Dog, *args)` *creates* the instance, then `instance.__init__(*args)` *initialises* it. `__init__` is what you normally write; `__new__` is for the rare cases where creation itself must be controlled — subclassing an immutable type (`int`, `str`, `tuple`) whose value is fixed at creation, or returning an existing object instead of a new one:

```
class Celsius(float):
    def __new__(cls, value):
        return super().__new__(cls, value)           # a float's value is set

class Registry:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance                          # a singleton: every cal
```

`__new__` is a static method receiving the class; if it returns an instance of `cls`, `__init__` runs on it; if it returns something else, `__init__` is skipped. The singleton above works, and a module-level instance is usually simpler — a module is already a singleton.

---

init_subclass

The hook for "run code when a class is subclassed" — registration, validation, filling in defaults — without a metaclass:

```
class Plugin:
    registry: dict[str, type] = {}

    def __init_subclass__(cls, *, key=None, **kwargs):
        super().__init_subclass__(**kwargs)
        key = key or cls.__name__.lower()
        if key in Plugin.registry:
            raise TypeError(f"duplicate plugin key {key!r}")
        Plugin.registry[key] = cls

class CsvExport(Plugin, key="csv"): ...
class JsonExport(Plugin): ...

Plugin.registry                    # {'csv': CsvExport, 'jsonexport': JsonExpo
```

It runs on the *base* class each time a subclass is created, receives the new class as `cls`, and can take keyword arguments from the `class` statement. Validation belongs here too: check that a subclass defines the attributes the base requires (`if not hasattr(cls, "name"): raise TypeError`) and the error appears at definition time, where a missing `@abstractmethod` would appear at instantiation.

---

class_getitem

`list[int]` works because `list` defines `__class_getitem__`, which receives the subscript and returns something (a `types.GenericAlias`) usable as a hint. `Generic[T]` supplies it for your classes (Module 15); defining it directly is for a class that wants `MyType[...]` to mean something of its own.

Metaclasses, in outline

A metaclass is a subclass of `type` used as the class of a class: `class Model(metaclass=ModelMeta)`. Its `__new__` runs when the `class` statement executes, receives the name, bases and namespace, and may rewrite them before the class exists — collect the fields declared in the body, generate methods, enforce naming rules, record definition order:

```
class ModelMeta(type):
    def __new__(mcls, name, bases, namespace, **kwargs):
        fields = [k for k, v in namespace.items() if isinstance(v, Field)]
        namespace["_fields"] = fields
        return super().__new__(mcls, name, bases, namespace)

class Model(metaclass=ModelMeta): ...
class User(Model):
    name = Field(str)
    age = Field(int)

User._fields                        # ['name', 'age']
```

Django's models, SQLAlchemy's declarative base, `enum.Enum` and `abc.ABC` use metaclasses. Almost nobody else should: `__init_subclass__` handles registration and validation, class decorators handle transformation, descriptors handle attribute behaviour — and a metaclass complicates every subclass and every combination with other metaclasses (two bases with

different metaclasses is a `TypeError`). The honest rule: if you can explain why `__init_subclass__` is not enough, you may write a metaclass.

---

prepare and namespace order

A metaclass may define `__prepare__` to return the mapping the class body is executed in — historically an `OrderedDict` to record field order. Since 3.7 a plain dict is ordered, so `namespace.items()` already reflects definition order, and `__prepare__` is a curiosity.

Creating and inspecting at run time

```
cls = type(obj)                    # the class of anything
issubclass(cls, Base)
cls.__mro__                        # the lookup order
inspect.isclass(x), inspect.getmembers(cls)
setattr(cls, "method", fn)         # monkey-patching: legal, useful in tests,
```

Every tool here is ordinary attribute access on an ordinary object; that is the point of "classes are objects".

Pitfalls

- A metaclass where `__init_subclass__` or a decorator would do.
- Returning a non-instance from `__new__` and expecting `__init__` to run.
- `__init__` on an immutable subclass trying to set the value (too late; use `__new__`).
- Forgetting `super().__init_subclass__(**kwargs)` in a hook that takes keywords.
- Two bases with different metaclasses.
- Monkey-patching a class outside a test and surprising every other user of it.

Key takeaways

- Classes are instances of `type`; `type(name, bases, namespace)` creates one from data; classes are mutable.

`__new__` creates, `__init__` initialises; override `__new__` for immutable subclasses and singletons only.

`__init_subclass__` runs on the base when a subclass is defined — the tool for registries and definition-time validation.

`__class_getitem__` is what makes `Cls[...]` legal; `Generic` provides it.

Metaclasses rewrite classes as they are created; frameworks use them, applications almost never should.

EXERCISE 1

A registry of dynamic plugins

`Plugin` keeps a class-level `registry` and an `__init_subclass__(cls, *, key=None, **kwargs)` that registers each subclass under `key` (default: the class name lower-cased), raising `TypeError("duplicate key <k>")` for a repeat. Commands: `plugin Name [key]` creates a subclass **at run time** with `type(Name, (Plugin,), {}, key=key)` (print the error message on a duplicate); `list` prints `key=ClassName` pairs in registration order; `is Name` prints whether the named class is a subclass of `Plugin`.

Input: commands. **Output:** the results of `list`, `is` and failed `plugin` commands.

```
plugin CsvExport csv
plugin JsonExport
plugin Other csv
list
is JsonExport
```

prints

```
duplicate key csv
csv=CsvExport jsonexport=JsonExport
True
```

INPUT 1

```
plugin CsvExport csv
plugin JsonExport
plugin Other csv
list
is JsonExport
```

OUTPUT 1

```
duplicate key csv
csv=CsvExport jsonexport=JsonExport
True
```

REFERENCE SOLUTION

```python
import sys


class Plugin:
    registry: dict[str, type] = {}

    def __init_subclass__(cls, *, key=None, **kwargs):
        super().__init_subclass__(**kwargs)
        key = key or cls.__name__.lower()
        if key in Plugin.registry:
            raise TypeError(f"duplicate key {key}")
        Plugin.registry[key] = cls


created = {}
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "plugin":
        name = args[0]
        key = args[1] if len(args) > 1 else None
        try:
            created[name] = type(name, (Plugin,), {}, key=key)
        except TypeError as e:
            print(e)
    elif cmd == "list":
        print(" ".join(f"{k}={v.__name__}" for k, v in Plugin.registry.items()))
    elif cmd == "is":
        print(issubclass(created[args[0]], Plugin))
```

EXERCISE 2

__new__: a singleton and an immutable subclass

Write `Registry` whose `__new__` returns one shared instance (created on first use) and `Celsius(float)` whose `__new__` builds the float and whose `fahrenheit` property converts. Commands: `registry` creates two registries and prints whether they are the same object; `celsius x` prints `<x> C = <f> F` with one decimal and whether the value is an instance of `float`; `init-count` prints how many times `Registry.__init__` has run in total (a class counter — note it runs on every construction even though `__new__` returns the same object).

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

```
registry
celsius 100
init-count
```

prints

```
same True
100.0 C = 212.0 F float True
inits 2
```

INPUT 1

```
registry
celsius 100
init-count
```

OUTPUT 1

```
same True
100.0 C = 212.0 F float True
inits 2
```

REFERENCE SOLUTION

```
import sys


class Registry:
    _instance = None
    inits = 0

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        type(self).inits += 1


class Celsius(float):
    def __new__(cls, value):
        return super().__new__(cls, value)

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


for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "registry":
        a, b = Registry(), Registry()
        print(f"same {a is b}")
    elif cmd == "celsius":
        c = Celsius(float(args[0]))
        print(f"{c:.1f} C = {c.fahrenheit:.1f} F float {isinstance(c, float)}")
    elif cmd == "init-count":
        print(f"inits {Registry.inits}")
```

QUIZ

1. What is `type(type)`?

A. `object`

B. `type` — it is its own class

C. `NoneType`

D. `class`

2. What does `type("Cat", (object,), {"legs": 4})` return?

A. An instance with 4 legs

B. A new class named `Cat` with a class attribute `legs`

C. A tuple

D. A dict

3. When does `__init__` *not* run after `__new__`?

A. Never

B. When `__new__` returns an object that is not an instance of the class

C. When the class has `__slots__`

D. When `__new__` is inherited

4. What is `__init_subclass__` for?

A. Initialising instances

B. Running code on the base class each time a subclass is defined — registries and validation

C. Replacing `__init__`

D. Creating metaclasses

5. When is a metaclass justified?

A. Whenever a class needs registering

B. Only when the class namespace itself must be rewritten as the class is created and `__init_subclass__` or a decorator cannot do it

C. For every base class

D. Never

ANSWER KEY · CLASSES AS OBJECTS — TYPE, __NEW__, __INIT_SUBCLASS__ AND METACLASSES IN OUTLINE

1. B `type` is the metaclass of every ordinary class, including itself.

2. B The `class` statement is a call to `type` with the name, bases and the body's namespace.

3. B For a singleton that returns the same instance, `__init__` runs every time — a common surprise.

4. B It replaces most reasons anyone wrote a metaclass.

5. B Frameworks (ORMs, `enum`, `abc`) use them; applications almost never need to.

LESSON 6 · 14 MIN

The data model — the rest of the dunders, and a Vector that uses them

Module 8 covered the dunders a value type needs first. This lesson completes the map: `__call__`, `__format__`, `__index__`, `__bool__`, `__contains__`, the in-place operators (`__iadd__`), unary and reflected arithmetic, `__matmul__`, `__missing__` for dict subclasses, `__reversed__`, `__del__` and why it cannot be relied on, and the dispatch rules — `NotImplemented`, reflected fallbacks, subclass priority — that make mixed-type operations work. It ends with a `Vector` class that exercises most of them, as the checkpoint will.

Calling, formatting, converting

```
class Multiplier:
    def __init__(self, k):
        self.k = k
    def __call__(self, x):                  # obj(x)
        return self.k * x

class Money:
    def __format__(self, spec):             # f"{m:>10}" — spec is what follow
        text = f"{self.cents / 100:.2f}"
        return format(text, spec)
    def __bool__(self):                     # truthiness; without it, __len__,
        return self.cents != 0
    def __int__(self): return self.cents // 100
    def __float__(self): return self.cents / 100
    def __index__(self): return self.cents  # lets the object be used as a sli
```

`__call__` makes instances callable — stateful callables, decorator classes, strategy objects. `__format__` receives the format spec and returns the formatted text; the default delegates to `__str__` and ignores the spec. `__index__` is the "this is really an integer" hook: `range`, slicing, `bin` and `hex` accept anything that defines it.

Arithmetic: forward, reflected, in-place

For `a + b`, Python tries `a.__add__(b)`; if that returns `NotImplemented`, it tries `b.__radd__(a)`; if both fail, `TypeError`. One exception: if `type(b)` is a *subclass* of `type(a)` and overrides `__radd__`, the reflected method is tried first, so a subclass can specialise mixed operations. `a += b` tries `a.__iadd__(b)` and falls back to `a = a + b`; an `__iadd__` mutates and must `return self`.

Operator	Forward	Reflected	In-place
`+` `-` `*` `/` `//` `%` `**`	`__add__` … `__pow__`	`__radd__` … `__rpow__`	`__iadd__` … `__ipow__`
`@`	`__matmul__`	`__rmatmul__`	`__imatmul__`
`&` `|` `^` `<<` `>>`	`__and__` … `__rshift__`	`__rand__` …	`__iand__` …
`-x` `+x` `abs(x)` `~x`	`__neg__` `__pos__` `__abs__` `__invert__`		
`round(x, n)` `math.floor(x)`	`__round__` `__floor__` `__ceil__` `__trunc__`		
`divmod(a, b)`	`__divmod__`	`__rdivmod__`	

Return `NotImplemented` (the singleton, not the exception) for operands you do not handle; never raise `TypeError` yourself, or the reflected attempt is skipped.

Containers

```
class Grid:
    def __getitem__(self, key):              # key may be an int, a slice, or a
        if isinstance(key, tuple):
            r, c = key
            return self._cells[r][c]
        return self._cells[key]
    def __setitem__(self, key, value): ...
    def __delitem__(self, key): ...
    def __contains__(self, value): ...       # in; falls back on iteration
    def __reversed__(self): ...              # reversed(); falls back on __len
    def __len__(self): ...
    def __iter__(self): ...

class Defaults(dict):
    def __missing__(self, key):              # called by dict.__getitem__ for a
        return f"<{key}>"
```

`g[1, 2]` passes the tuple `(1, 2)` as one key — the syntax NumPy uses. `__missing__` is the hook `defaultdict` is built on: `dict[key]` calls it instead of raising; `get`, `in` and `setdefault` do not.

Object lifetime

`__del__` runs when the reference count reaches zero — usually promptly in CPython, never guaranteed (cycles, interpreter shutdown, other implementations), and exceptions in it are printed and ignored. It is not a destructor to rely on: resources are released by `with` and `close()`, not by `__del__`. `weakref.finalize` is the reliable way to attach cleanup to an object's collection when you must.

A Vector

```python
import math

class Vector:
    __slots__ = ("_xs",)

    def __init__(self, *xs):
        self._xs = tuple(float(x) for x in xs)

    def __repr__(self):
        return f"Vector({', '.join(f'{x:g}' for x in self._xs)})"

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

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

    def __getitem__(self, i):
        result = self._xs[i]
        return Vector(*result) if isinstance(i, slice) else result

    def __eq__(self, other):
        return isinstance(other, Vector) and self._xs == other._xs

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

    def __bool__(self):
        return any(self._xs)

    def __abs__(self):
        return math.hypot(*self._xs)

    def __neg__(self):
        return Vector(*(-x for x in self._xs))

    def __add__(self, other):
        if not isinstance(other, Vector) or len(other) != len(self):
            return NotImplemented
        return Vector(*(a + b for a, b in zip(self, other)))

    def __mul__(self, k):                    # scalar
        if not isinstance(k, (int, float)):
            return NotImplemented
        return Vector(*(x * k for x in self._xs))
```

```
    __rmul__ = __mul__                          # 3 * v

    def __matmul__(self, other):                # dot product
        if not isinstance(other, Vector) or len(other) != len(self):
            return NotImplemented
        return sum(a * b for a, b in zip(self, other))

    def __format__(self, spec):
        return f"({', '.join(format(x, spec) for x in self._xs)})"
```

`Vector(1, 2) + Vector(3, 4)`, `3 * v`, `v @ w`, `abs(v)`, `-v`, `v[0]`, `v[1:]`, `len(v)`, `list(v)`, `bool(Vector(0, 0))`, `{v}`, `f"{v:.1f}"` — each is one dunder, and together they make the class feel built in. `__slots__` keeps instances small; `__eq__` with `__hash__` makes them set members; every binary operator returns `NotImplemented` for foreign types.

Pitfalls

Raising `TypeError` instead of returning `NotImplemented`.

`__iadd__` that mutates but forgets `return self`.

`__format__` ignoring the spec, or `__str__` when `__format__` was needed for `f"{x:>8}"`.

Relying on `__del__` to release a resource.

`__getitem__` that handles ints but not slices (or the reverse).

`__hash__` on a mutable object.

Key takeaways

`__call__`, `__format__`, `__bool__`, `__index__`, `__int__` / `__float__` connect an object to calling, f-strings, truth and integer contexts.

Binary operators try the forward method, then the reflected one; return `NotImplemented`, not an exception; `__iadd__` returns `self`.

`__getitem__` receives ints, slices or tuples; `__missing__` customises dict misses; `__contains__` and `__reversed__` have iteration fallbacks.

`__del__` is not a reliable destructor; use `with`, `close` or `weakref.finalize`.

A complete value type is a dozen small methods; the checkpoint's `Vector` is the template.

EXERCISE 1

Vector arithmetic

Implement `Vector(*xs)` with `__repr__` (`Vector(1, 2)` using `:g`), `__len__`, `__iter__`, `__eq__`, `__add__`, `__mul__` and `__rmul__` (scalar), `__matmul__` (dot product), `__neg__`, `__abs__` and `__format__` (applying the spec to each component inside parentheses). Binary operators return `NotImplemented` for foreign operands. Read `v` and `w` as two lines of numbers, then commands `add`, `dot`, `scale k`, `rscale k` (`k * v`), `neg`, `abs`, `eq`, `fmt <spec>`, and `bad` (which evaluates `v + 1` and prints the exception type name).

Input: two lines of numbers, then commands. **Output:** one line per command.

```
1 2
3 4
add
dot
rscale 3
fmt .1f
bad
```

prints

```
Vector(4, 6)
11
Vector(3, 6)
(1.0, 2.0)
TypeError
```

INPUT 1

```
1 2
3 4
add
dot
rscale 3
fmt .1f
bad
```

OUTPUT 1

```
Vector(4, 6)
11
Vector(3, 6)
(1.0, 2.0)
TypeError
```

REFERENCE SOLUTION

```
import math
import sys


class Vector:
    def __init__(self, *xs):
        self._xs = tuple(float(x) for x in xs)

    def __repr__(self):
        return f"Vector({', '.join(f'{x:g}' for x in self._xs)})"

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

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

    def __eq__(self, other):
        return isinstance(other, Vector) and self._xs == other._xs

    def __add__(self, other):
        if not isinstance(other, Vector) or len(other) != len(self):
            return NotImplemented
        return Vector(*(a + b for a, b in zip(self, other)))

    def __mul__(self, k):
        if not isinstance(k, (int, float)):
            return NotImplemented
        return Vector(*(x * k for x in self._xs))

    __rmul__ = __mul__

    def __matmul__(self, other):
        if not isinstance(other, Vector) or len(other) != len(self):
            return NotImplemented
        return sum(a * b for a, b in zip(self, other))

    def __neg__(self):
        return Vector(*(-x for x in self._xs))

    def __abs__(self):
        return math.hypot(*self._xs)

    def __format__(self, spec):
        return f"({', '.join(format(x, spec) for x in self._xs)})"


v = Vector(*map(float, input().split()))
w = Vector(*map(float, input().split()))
```

```
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "add":
        print(v + w)
    elif cmd == "dot":
        print(f"{v @ w:g}")
    elif cmd == "scale":
        print(v * float(args[0]))
    elif cmd == "rscale":
        print(float(args[0]) * v)
    elif cmd == "neg":
        print(-v)
    elif cmd == "abs":
        print(f"{abs(v):.2f}")
    elif cmd == "eq":
        print(v == w)
    elif cmd == "fmt":
        print(f"{v:{args[0]}}")
    elif cmd == "bad":
        try:
            v + 1
        except TypeError as e:
            print(type(e).__name__)
```

EXERCISE 2

A grid with tuple keys and a forgiving dict

Write `Grid(rows, cols)` storing cells in a flat list with `__getitem__` / `__setitem__` taking a `(r, c)` tuple (`IndexError` when out of range) and `__contains__` testing whether a value is stored anywhere; and `Defaults(dict)` with a `__missing__` returning `<key>` in angle brackets without inserting it. Commands: `set r c v`, `get r c` (print value or `out of range`), `has v`, `lookup k` (on a `Defaults` built from the first line's pairs), `size` (the dict's length, to show `__missing__` inserted nothing).

Input: a line of `key=value` pairs, then `R C`, then commands. **Output:** one line per query.

```
a=1
2 2
set 0 1 7
get 0 1
get 5 5
has 7
lookup a
lookup z
size
```

prints

```
7
out of range
True
1
<z>
1
```

INPUT 1

```
a=1
2 2
set 0 1 7
get 0 1
get 5 5
has 7
lookup a
lookup z
size
```

OUTPUT 1

```
7
out of range
True
1
<z>
1
```

REFERENCE SOLUTION

```
import sys


class Grid:
    def __init__(self, rows, cols):
        self.rows, self.cols = rows, cols
        self._cells = [0] * (rows * cols)

    def _index(self, key):
        r, c = key
        if not (0 <= r < self.rows and 0 <= c < self.cols):
            raise IndexError(key)
        return r * self.cols + c

    def __getitem__(self, key):
        return self._cells[self._index(key)]

    def __setitem__(self, key, value):
        self._cells[self._index(key)] = value

    def __contains__(self, value):
        return value in self._cells


class Defaults(dict):
    def __missing__(self, key):
        return f"<{key}>"


defaults = Defaults(tok.split("=", 1) for tok in input().split())
rows, cols = map(int, input().split())
grid = Grid(rows, cols)
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "set":
        grid[int(args[0]), int(args[1])] = int(args[2])
    elif cmd == "get":
        try:
            print(grid[int(args[0]), int(args[1])])
        except IndexError:
            print("out of range")
    elif cmd == "has":
        print(int(args[0]) in grid)
    elif cmd == "lookup":
        print(defaults[args[0]])
    elif cmd == "size":
        print(len(defaults))
```

QUIZ

1. For `a + b`, what does Python try after `a.__add__(b)` returns `NotImplemented`?

A. Raises `TypeError` immediately

B. `b.__radd__(a)`

C. `a.__iadd__(b)`

D. `b.__add__(a)`

2. What must `__iadd__` return?

A. `None`

B. `self` (after mutating)

C. A new object

D. `NotImplemented` always

3. What does `g[1, 2]` pass to `__getitem__`?

A. Two arguments

B. The tuple `(1, 2)` as one key

C. A slice

D. `1` only

4. Which dict operation calls `__missing__`?

A. `d.get(k)`

B. `d[k]` for an absent key

C. `k in d`

D. `d.setdefault(k)`

5. Why is `__del__` not a reliable place to release a resource?

A. It is never called

B. It runs only when the last reference disappears, which cycles and shutdown can delay or skip; use `with` or `close`

C. It cannot access `self`

D. It raises on error

ANSWER KEY · THE DATA MODEL — THE REST OF THE DUNDERS, AND A VECTOR THAT USES THEM

1. B The reflected method gets a chance; only if both decline is `TypeError` raised.

2. B `a += b` rebinds `a` to whatever `__iadd__` returns.

3. B Commas in a subscript build a tuple; NumPy indexing works this way.

4. B Only subscripting; `get`, `in` and `setdefault` bypass it.

5. B Deterministic cleanup belongs to context managers; `weakref.finalize` is the fallback.

CHECKPOINT · 25 MIN

Checkpoint — Decorators, descriptors and the data model

This checkpoint covers the whole module: decorators as `f = deco(f)` with `@wraps`, arguments and stacking, closures with cells, late binding and factories, the descriptor protocol behind properties, methods and validated attributes, the attribute hooks `__getattr__`, `__getattribute__` and `__setattr__` with reflection and `__slots__`, classes as objects with `type`, `__new__`, `__init_subclass__` and metaclasses, and the rest of the data model through a `Vector`.

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 `@retry(3)` above a `def` expand to, and how many nested functions does `retry` need?
- Why do three lambdas created in a loop all see the same `i`, and what are the two fixes?
- What is the difference between a data and a non-data descriptor, and which one is `property`?
- When is `__getattr__` called, and what must it raise for names it does not handle?
- What does `__init_subclass__` receive, and what is it for?
- What should a binary dunder return for an operand it does not understand?
- Why must `__setattr__` write through `object.__setattr__`?

The three programs are a decorator suite — timing-free tracing, counting and memoising — stacked in both orders, a small validation framework built from descriptors with `__set_name__` and a registering `__init_subclass__`, and a complete `Vector` value type exercised through arithmetic, formatting and containment.

EXERCISE 1

Stacked both ways

Write `traced` (prints `call <name>(<n>)` on each invocation) and `memoized` (caches by argument), both with `@wraps`. Define `fib` twice: `fib_a` as `@traced` over `@memoized`, and `fib_b` as `@memoized` over `@traced`; each computes Fibonacci recursively through its own decorated name. Read `n`; call `fib_a(n)`, count the trace lines it printed, then `fib_b(n)`, count again, and print `a traced <x>`, `b traced <y>`, `result <fib(n)>`. (Capture the traces with `contextlib.redirect_stdout` into a `StringIO` and count its lines.)

Input: `n`. **Output:** three lines.

```
5
```

prints

```
a traced 9
b traced 6
result 5
```

INPUT 1

```
5
```

OUTPUT 1

```
a traced 9
b traced 6
result 5
```

REFERENCE SOLUTION

```
import contextlib
import io
from functools import wraps


def traced(fn):
    @wraps(fn)
    def wrapper(n):
        print(f"call {fn.__name__}({n})")
        return fn(n)
    return wrapper


def memoized(fn):
    cache = {}

    @wraps(fn)
    def wrapper(n):
        if n not in cache:
            cache[n] = fn(n)
        return cache[n]
    return wrapper


@traced
@memoized
def fib_a(n):
    return n if n < 2 else fib_a(n - 1) + fib_a(n - 2)


@memoized
@traced
def fib_b(n):
    return n if n < 2 else fib_b(n - 1) + fib_b(n - 2)


def count_traces(fn, n):
    buf = io.StringIO()
    with contextlib.redirect_stdout(buf):
        result = fn(n)
    return result, len(buf.getvalue().splitlines())


n = int(input())
result_a, traces_a = count_traces(fib_a, n)
result_b, traces_b = count_traces(fib_b, n)
print(f"a traced {traces_a}")
```

```
print(f"b traced {traces_b}")
print(f"result {result_a}")
```

EXERCISE 2

A small validation framework

Build `Field`, a descriptor with `__set_name__`, `__get__` and `__set__` that checks `isinstance(value, self.kind)` and raises `TypeError("<name> expects <kind>, got <type>")`; and `Model`, whose `__init_subclass__` records the subclass's `Field` attributes in `cls.fields` (names in definition order) and whose `__init__(**kwargs)` assigns each field from the keywords (missing → `TypeError("missing <name>")`). Two models are given: `User(name=str, age=int)` and `Point(x=float, y=float)`. Commands: `new Model k=v …` (values parsed with `ast.literal_eval`; print `ok <repr>` or the error message) and `fields Model`.

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

```
fields User
new User name="ada" age=36
new User name="bob" age="x"
new Point x=1.5
```

prints

```
name age
ok User(name='ada', age=36)
age expects int, got str
missing y
```

INPUT 1

```
fields User
new User name="ada" age=36
new User name="bob" age="x"
new Point x=1.5
```

OUTPUT 1

```
name age
ok User(name='ada', age=36)
age expects int, got str
missing y
```

REFERENCE SOLUTION

```
import ast
import sys


class Field:
    def __init__(self, kind):
        self.kind = kind

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

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

    def __set__(self, obj, value):
        if not isinstance(value, self.kind):
            raise TypeError(f"{self.name} expects {self.kind.__name__}, got {type(va
        setattr(obj, self.storage, value)


class Model:
    fields: tuple[str, ...] = ()

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.fields = tuple(k for k, v in cls.__dict__.items() if isinstance(v, Field

    def __init__(self, **kwargs):
        for name in self.fields:
            if name not in kwargs:
                raise TypeError(f"missing {name}")
            setattr(self, name, kwargs[name])

    def __repr__(self):
        inner = ", ".join(f"{f}={getattr(self, f)!r}" for f in self.fields)
        return f"{type(self).__name__}({inner})"


class User(Model):
    name = Field(str)
    age = Field(int)


class Point(Model):
    x = Field(float)
    y = Field(float)
```

```
MODELS = {"User": User, "Point": Point}
for line in sys.stdin:
    cmd, model, *pairs = line.split()
    cls = MODELS[model]
    if cmd == "fields":
        print(" ".join(cls.fields))
    elif cmd == "new":
        kwargs = {k: ast.literal_eval(v) for k, v in (p.split("=", 1) for p in pairs
        try:
            print(f"ok {cls(**kwargs)!r}")
        except TypeError as e:
            print(e)
```

EXERCISE 3

The complete Vector

Extend the `Vector` of lesson 6 with `__getitem__` (an int returns a component; a slice returns a `Vector`), `__hash__` consistent with `__eq__`, `__bool__` (any non-zero component), `__sub__`, and `__iadd__` that mutates in place and returns `self`. Read `v`, then commands: `get i`, `slice a b`, `bool`, `sub x y …` (prints `v - Vector(...)`), `iadd x y …` (applies `+=` and prints `v` and whether it is still the same object as before), `hash-eq x y …` (prints whether `v == Vector(...)` and whether their hashes agree).

Input: a line of numbers, then commands. **Output:** one line per command.

```
1 2 3
get 0
slice 1 3
bool
sub 1 1 1
iadd 1 1 1
hash-eq 2 3 4
```

prints

```
1.0
Vector(2, 3)
True
Vector(0, 1, 2)
Vector(2, 3, 4) same True
eq True hash True
```

INPUT 1

```
1 2 3
get 0
slice 1 3
bool
sub 1 1 1
iadd 1 1 1
hash-eq 2 3 4
```

OUTPUT 1

```
1.0
Vector(2, 3)
True
Vector(0, 1, 2)
Vector(2, 3, 4) same True
eq True hash True
```

REFERENCE SOLUTION

```
import math
import sys


class Vector:
    def __init__(self, *xs):
        self._xs = list(float(x) for x in xs)

    def __repr__(self):
        return f"Vector({', '.join(f'{x:g}' for x in self._xs)})"

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

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

    def __eq__(self, other):
        return isinstance(other, Vector) and self._xs == other._xs

    def __getitem__(self, i):
        if isinstance(i, slice):
            return Vector(*self._xs[i])
        return self._xs[i]

    def __hash__(self):
        return hash(tuple(self._xs))

    def __bool__(self):
        return any(self._xs)

    def __sub__(self, other):
        if not isinstance(other, Vector) or len(other) != len(self):
            return NotImplemented
        return Vector(*(a - b for a, b in zip(self, other)))

    def __iadd__(self, other):
        if not isinstance(other, Vector) or len(other) != len(self):
            return NotImplemented
        for i, b in enumerate(other):
            self._xs[i] += b
        return self


v = Vector(*map(float, input().split()))
for line in sys.stdin:
    cmd, *args = line.split()
    nums = [float(a) for a in args]
    if cmd == "get":
```

```
        print(v[int(args[0])])
    elif cmd == "slice":
        print(v[int(args[0]):int(args[1])])
    elif cmd == "bool":
        print(bool(v))
    elif cmd == "sub":
        print(v - Vector(*nums))
    elif cmd == "iadd":
        before = v
        v += Vector(*nums)
        print(f"{v} same {v is before}")
    elif cmd == "hash-eq":
        other = Vector(*nums)
        print(f"eq {v == other} hash {hash(v) == hash(other)}")
```

CHECKPOINT QUESTIONS

1. When does the decorator function itself run?

A. On every call

B. Once, when the `def` is executed

C. At import of `functools`

D. Never

2. What does `functools.wraps` set on the wrapper?

A. Only `__name__`

B. `__name__`, `__doc__`, `__qualname__`, `__module__` and `__wrapped__`

C. The call count

D. The signature only

3. Why do closures in a loop all return the final loop value?

A. A compiler bug

B. They share one cell for the loop variable, read when called

C. Lambdas cannot capture

D. The loop copies the value

4. Which method makes a descriptor a *data* descriptor?

A. `__get__`

B. `__set__` or `__delete__`

C. `__set_name__`

D. `__init__`

5. Where should a descriptor store per-instance values?

A. On the descriptor

B. On the instance, under a private name

C. In a global dict

D. In the class

6. What does `object.__setattr__(self, name, value)` do inside a custom `__setattr__`?

A. Nothing

B. Performs the real assignment without re-entering the custom hook

C. Raises

D. Calls `__getattr__`

7. What is `hasattr(obj, "x")` implemented as?

A. A dict lookup

B. `getattr` inside a `try`, returning `False` on `AttributeError`

C. `"x" in dir(obj)`

D. A metaclass call

8. What does `type(name, bases, ns)` need to be told keyword arguments for?

A. Nothing

B. They are forwarded to `__init_subclass__` of the bases

C. The metaclass name

D. Instance defaults

9. What does returning `NotImplemented` from `__eq__` cause?

A. A `TypeError`

B. Python tries the reflected comparison and finally falls back to identity

C. `False`

D. `None`

10. Which pair must agree for an object to be a set member?

A. `__repr__` and `__str__`

B. `__eq__` and `__hash__`

C. `__lt__` and `__gt__`

D. `__len__` and `__bool__`

11. What is `v[1:3]` if `__getitem__` receives a `slice` and returns `Vector(*self._xs[i])`?

A. A list

B. A `Vector` of the two components

C. A tuple

D. `IndexError`

12. Which tool should register subclasses without a metaclass?

A. `__new__`

B. `__init_subclass__`

C. `__getattr__`

D. `__call__`

ANSWER KEY · CHECKPOINT — DECORATORS, DESCRIPTORS AND THE DATA MODEL

1. B The wrapper it returns is what runs on each call.

2. B `__wrapped__` lets tests and tools reach the original.

3. B Late binding: bind per iteration with a default argument or a factory.

4. B Data descriptors take priority over the instance dict.

5. B The descriptor object is shared by every instance of the class.

6. B It is the way to avoid infinite recursion in write hooks.

7. B That is why `__getattr__` must raise `AttributeError` rather than return `None`.

8. B `class X(Base, key="csv")` and `type("X", (Base,), {}, key="csv")` are equivalent.

9. B It is the cooperative signal; raising would break the protocol.

10. B Equal objects must hash equal; defining `__eq__` alone removes the hash.

11. B Handling `slice` explicitly is what keeps slicing inside the type.

12. B It runs on the base for each new subclass and can take class-statement keywords.