第一节：对象模型与函数机制

1. 对象与类型：一切皆对象

Python 中所有数据都是对象，每个对象有三个基本属性：

- 身份（id）：创建后不变，可用 id() 查看，可近似看作内存地址。
- 类型（type）：创建后不变，决定对象支持哪些操作。
- 值（value）：可变对象的值可改变，不可变对象的值不能改变。

2. 可变类型 vs 不可变类型（面试必考）

不可变类型	可变类型
int, float, bool, str, tuple, frozenset, bytes	list, dict, set, bytearray

不可变对象：一旦创建，值就不能修改。任何看似“修改”的操作，本质都是创建新对象并返回。

```
s = "hello"
s.upper()          # 创建新字符串 "HELLO"，s 本身没变
s = s + "!"        # 创建了新字符串，s 标签贴到新对象上
```

可变对象：值可以在原地修改，id 不变。

```
a = [1,2,3]
a.append(4)      # a 的内容变了，但 id(a) 不变
```

3. 变量只是“标签”（引用）

- 变量不是盒子，是指向对象的引用。赋值语句 x = 10 是把标签 x 贴到整数对象 10 上。
- 多变量可以指向同一对象。a = [1,2]; b = a 则 a 和 b 贴到同一个列表对象，通过任一标签修改都会影响另一个。
- 重新赋值只是撕下标签贴到别处：b = [3,4] 不影响 a。

4. == 与 is 的区别

- == 比较值是否相等，会调用 __eq__ 魔法方法。
- is 比较身份是否相同，即 id(a) == id(b)。只有同一个对象才会 True。
- 典型面试题：a = 256; b = 256; a is b 为何是 True？Python 对小整数（-5~256）有缓存，指向同一对象。a = 257; b = 257; a is b 通常为 False（在交互环境不同语句）。

5. 函数参数的传递方式：传对象引用（Pass by Assignment）

这是面试的重灾区。Python 既不是传值也不是传引用，而是传对象引用，也叫“传对象，但引用是按值传递的”。

规则：函数传入的是对象引用（形参贴到实参所指的对象上），在函数内部：

- 如果修改可变对象的内容（如 append、修改属性），会影响外部对象。
- 如果对形参重新赋值（x = 新对象），只会让形参标签指向新对象，不影响外部变量。

示例：

```
def f(lst):
    lst.append(4)       # 修改了传入的列表对象

a = [1,2,3]
f(a)
print(a)                # [1,2,3,4]  影响了

def g(x):
    x = 2               # 只是形参重新指向对象 2，外部不变

b = 1
g(b)
print(b)                # 1
```

核心记忆法：形参和实参最初指向同一个对象；在内部“改内容”会影响外部（前提是可变类型），“改指向”不

会影响外部。

6. 默认参数的陷阱

```
def buggy(lst=[]):
    lst.append(1)
    return lst

print(buggy())    # [1]
print(buggy())    # [1,1]  不是新列表!
```

默认参数在函数定义时只计算一次，之后每次调用如果没有传参，用的都是同一个对象（那块内存）。所以默认值一定要用不可变对象，或者用 None 判空再初始化：

```
def safe(lst=None):
    if lst is None:
        lst = []
    lst.append(1)
    return lst
```

7. 作用域 LEGB 规则

变量查找顺序：Local -> Enclosing -> Global -> Built-in

- Local：函数内部定义的变量
- Enclosing：外部嵌套函数的作用域（闭包相关）
- Global：模块级别的全局变量
- Built-in：Python 内置的名字（print, len 等）

要在函数内修改全局变量需 global 声明；修改嵌套作用域变量需 nonlocal。

8. 闭包与 nonlocal

闭包：内部函数引用了外部函数作用域里的变量，并且外部函数已执行完毕，但内部函数仍能记住那个变量。

```
def outer(x):
    def inner(y):
        return x + y
    return inner
add5 = outer(5)    # x=5
print(add5(3))     # 8
```

这里 inner 是闭包，它记住了 x=5。如果内部函数要修改外部变量，必须用 nonlocal 声明，否则 Python 会认为 x 是 local 的（因为赋值），导致 UnboundLocalError。

9. 参数类型：位置、关键字、*args、**kwargs

- 位置参数：按顺序传入
- 关键字参数：func(a=1, b=2)
- 默认参数：def f(x=0)
- 可变位置参数：*args 接收多余的位置参数，打包成元组
- 可变关键字参数：**kwargs 接收多余的关键字参数，打包成字典
- 混合顺序：def f(pos, *args, kw=0, **kwargs)，调用时位置参数先，关键字参数后，且关键字参数必须在位置参数之后。

注意：Python 对小整数（默认 **-5 到 256**）有驻留机制，这个范围内的整数会被缓存，所以 a=256; b=256 时 a is b 是 True。但 **257 超出范围**，每次赋值都会创建新整数对象，因此 c is d 是 False。

要区分：

```
a=1                    a=500
b=a                    b=500
a is b                 a is b

e = [1,2]
```

```
f = [1,2]
e is f
```

e 和 f 并不是引用同一个对象，两者都创建了具有相同值的不同列表

y += [3]它直接修改 y 所指向的列表对象，**不创建新对象；+= 对于列表是原地操作**

要注意：返回值如果是函数名，没有()，就不是返回调用函数的结果，而是返回函数

第二节：函数进阶与类基础

1. lambda 匿名函数

语法：lambda 参数：表达式

lambda 的本质是一个**只能写单表达式的匿名函数**，返回值就是表达式的计算结果。

```
add = lambda x, y: x + y
print(add(3, 5))  # 8
```

使用场景：作为 sorted、map、filter 等函数的短小回调

```
students = [('Alice', 22), ('Bob', 19)]
students.sort(key=lambda s: s[1])  # 按年龄排序
```

限制：不能包含语句（如 if 语句体）、不能有 return、不能有多行逻辑。复杂逻辑必须用 def。

注意：lambda 也是闭包，会捕获外部变量，延迟绑定时有坑：

```
funcs = [lambda: i for i in range(3)]
print([f() for f in funcs])  # [2, 2, 2] 不是 [0, 1, 2]
```

因为 i 是自由变量，lambda 在调用时才去取值，此时循环已结束 i=2。修正法：lambda i=i: i 用默认参数绑定当前值。

funcs = [lambda i=i: i for i in range(3)]虽然 lambda 在调用前不执行，但是 for i in range(3)，会执行，结果就是 funcs = [lambda i=i: i for i in range(3)]执行完后，三个不执行的引用函数，引用对象是同一个执行完值为 2 的 for 循环语句运行后的结果

重要：函数定义时不执行，调用时才执行，调用前，函数放在任何类型里面都是一个函数对象，可杯调用

Python 中，函数本身也是对象，可以像整数、字符串一样存储在列表中，这句话要理解

2. 装饰器（Decorator）

装饰器本质是一个**接受函数作为参数、返回新函数**的可调用对象，用来在不修改原函数代码的情况下增加功能。

闭包实现装饰器：

```
def timer(func):
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} 耗时: {time.time()-start:.4f}s")
        return result
    return wrapper

@timer
def slow_func():
    time.sleep(0.1)
slow_func()  # 自动计时
```

@timer 等价于 slow_func = timer(slow_func)。这一句理解很重要，仅@timer 这个东西就会执行 timer 函数，

传入参数就是 slow_func 函数
timer 函数这时会被执行，但是 wrapper 只会被定义，不会被调用，返回 wrapper 函数的引用给 slow_func，**此时**：slow_func 这个名字不再指向原来的函数，而是指向 wrapper 函数。原来的 slow_func 被保存在 wrapper 的闭包变量 func 中，不再直接可访问。
Wrapper 内的 func 函数会变成修改前的 slow_func 函数，也就是休息 0.1 秒

关键细节：

- wrapper 必须使用 *args, **kwargs 保证通用性。
- 装饰后原函数元信息（__name__, __doc__）会变成 wrapper 的，需要用 functools.wraps 修复：（重要）

```
from functools import wraps
def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        ...
    return wrapper
```

带参数装饰器：再包一层函数接收参数，返回真正的装饰器：

```
def repeat(n):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(n):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello {name}")
```

执行时机：装饰器在**函数定义时**立即执行，不是在调用时。
装饰器传入参数：要实现带参数的装饰器，比如 @retry(times=5)，你需要**再嵌套一层函数**

```
from functools import wraps

def retry(times=3):                          # 外层：接收参数
    def decorator(func):                     # 中层：真正的装饰器
        @wraps(func)
        def wrapper(*args, **kwargs):        # 内层：包装函数
            last_exc = None
            for attempt in range(times):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    last_exc = e
            raise last_exc
        return wrapper
    return decorator

# 使用
@retry(times=5)
```

```
def unstable_func():
    print("尝试...")
    raise ValueError("失败")

unstable_func()
```

3. 生成器（Generator）与 yield

生成器函数：使用 yield 代替 return 的函数，调用时返回一个生成器对象，支持惰性迭代。外层函数接收装饰器的参数，返回一个真正的装饰器（即内层函数）

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for num in countdown(3):
    print(num)   # 3, 2, 1
```

这里 for 自动会调用 next 循环输出

yield 与 return 的核心区别：

return	yield
函数执行结束，返回一个值	函数暂停，返回一个值并保留当前状态
调用后状态丢失	下次 next() 时从暂停处继续执行
一次调用一个结果	可产生一个序列，一个一个产出

生成器表达式：类似列表推导式，但用小括号，惰性求值。

```
gen = (x**2 for x in range(10**6))   # 不会立即创建百万个元素
```

重要方法：

- next(gen)：获取下一个值，耗尽时抛出 StopIteration
- gen.send(value)：向生成器内部发送值，value 成为 yield 表达式的结果
- gen.close()：在生成器内部抛出 GeneratorExit

4. 类、实例、self

```
class Dog:
    species = "犬科"              # 类变量（类属性）

    def __init__(self, name):   # 初始化实例
        self.name = name          # 实例变量（实例属性）

    def bark(self):               # 实例方法
        print(f"{self.name} 汪汪叫")
```

self：代表实例本身。Python 隐式将调用者传给方法的第一个参数。d.bark() 等价 Dog.bark(d)。

__init__：不是构造函数，是**初始化方法**。真正的构造函数是 __new__（极少重写）。

类变量 vs 实例变量：

- 类变量由所有实例共享，通过 类名.变量 或 self.__class__.变量 访问和修改。
- 实例变量每个实例独有。

经典陷阱：

```
class Dog:
    tricks = []             # 类变量
```

```
    def add_trick(self, trick):
        self.tricks.append(trick)  # 修改了共享的类变量列表!

d1 = Dog(); d2 = Dog()
d1.add_trick("打滚")
print(d2.tricks)  #["打滚"]—— d2 也被影响了!
```

- **属性查找顺序**：实例 __dict__ → 类 __dict__ → 父类 MRO 链。读属性时会顺着找，但**通过实例赋值属性**永远是往实例 __dict__ 里写，哪怕类有同名变量也不影响类。

5. 魔法方法（常用 5 个）

方法	作用	调用时机
__init__(self)	初始化实例	obj = Class()
__str__(self)	用户友好字符串	print(obj), str(obj)
__repr__(self)	开发者调试信息	repr(obj)，交互环境直接输入变量
__call__(self)	让实例可调用	obj()
__getitem__(self, key)	索引访问	obj[key]

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __str__(self):
        return f"Point({self.x}, {self.y})"
    def __repr__(self):
        return f"Point({self.x!r}, {self.y!r})"
    def __call__(self):
        return self.x ** 2 + self.y ** 2
    def __getitem__(self, index):
        return (self.x, self.y)[index]
```

6. 继承、super、MRO

基本继承：

```
class Animal:
    def speak(self):
        return "..."

class Cat(Animal):
    def speak(self):
        return "喵喵"
```

super()：调用父类方法，在多继承中按照 **MRO（Method Resolution Order，方法解析顺序）** 查找下一个类。

```
class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)  # 调用 Animal.__init__
        self.color = color
```

MRO（C3 线性化算法）：

- 可通过 类名.__mro__ 或 类名.mro() 查看。

- 经典菱形继承：

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
print(D.__mro__)   # D -> B -> C -> A -> object
```

- 核心原则：子类在前，父类在后；保持定义时的顺序；每个类只出现一次。
- **super() 在 MRO 中的作用**：不是直接调用父类，而是沿着 MRO 链找到下一个类的方法，确保合作式多继承能正确协作。

7. @staticmethod 与 @classmethod

```
class MyClass:
    class_var = 0

    def instance_method(self):
        return "需要 self，操作实例"

    @classmethod
    def class_method(cls):
        return f"类方法，可操作 cls.class_var={cls.class_var}，常用于工厂方法"

    @staticmethod
    def static_method():
        return "静态方法，既不需要 self 也不需要 cls，纯粹的工具函数"
```

面试常见追问：为什么不用普通函数而要写 @staticmethod？——原因是组织代码，让函数在命名空间上归属于类，增加可读性。

特征	实例方法	@classmethod	@staticmethod
第一参数	self（实例）	cls（类）	无
能访问类变量	可以	可以	不能直接访问（需类名）
能访问实例变量	可以	不能	不能
典型用途	操作实例数据	工厂方法、修改类状态	工具函数，归属类下

第三节：内存管理、拷贝、内置函数、异常与上下文管理

1. Python 内存管理：引用计数与垃圾回收

- **引用计数**：每个对象内部维护一个计数，当计数归零时立即回收内存。
  - 增加引用：赋值、传参、放入容器
  - 减少引用：del、变量重新赋值、离开作用域（函数运行结束，局部变量销毁，引用减 1）

当一个对象的引用计数降为 0 时，它的 __del__ 会被调用（如果定义了），然后内存立即被回收。

- **循环引用问题**：两个对象互相引用，引用计数永远不为零。

```
a = []; b = []; a.append(b); b.append(a)   # 循环引用
```

Python 如何解决循环引用？

Python 采用 **辅助垃圾回收器（GC）** 来处理引用计数无法回收的循环引用。主要机制：

- **标记-清除（Mark & Sweep）**：定期扫描所有对象，标记从根集（全局变量、栈）可到达的对象，清除

不可达的循环垃圾。

- **分代回收（Generational GC）**：对象按生存时间分 0/1/2 代，新生对象在 0 代，存活越久代数越高，回收频率越低。可以用 gc 模块手动控制。

面试常见题：解释循环引用和 Python 如何解决，del 做了什么。

2. 浅拷贝 vs 深拷贝

- **浅拷贝** copy.copy(obj)：创建新容器对象，但里面的元素**引用相同**。内部对象还是原来的对象。
- **深拷贝** copy.deepcopy(obj)：递归拷贝整个对象树，完全独立。新对象

```
import copy
a = [[1,2], [3,4]]
b = copy.copy(a)          # 外层新列表，内层还是同一列表
c = copy.deepcopy(a)    # 完全独立

a[0][0] = 99
print(b[0])    # [99, 2] —— b 受影响
print(c[0])    # [1, 2]   —— c 不受影响
```

记忆：浅拷贝“只深一层”，深拷贝“递归复制”。

结合可变/不可变类型理解

- 如果列表中全是不可变对象（如整数、字符串），浅拷贝和深拷贝行为一样，因为不可变对象不能原地修改，你不会“改内容”干扰别人。
- 如果列表中有可变对象，浅拷贝会导致内外共享，修改内容会相互影响，这时才需要用深拷贝。

3. 常用内置高阶函数

函数	作用	返回值
map(fn, iterable)	对每个元素应用 fn	迭代器
filter(fn, iterable)	保留 fn 返回 True 的元素	迭代器
reduce(fn, iterable[, init])	累积计算	最终值（functools）
sorted(iterable, key=, reverse=)	排序	列表
zip(*iterables)	拉链配对	迭代器
enumerate(iterable, start=0)	带索引迭代	迭代器

当你**显式地像调用函数一样使用实例**时（即 instance()），才会自动调用 __call__

4. 异常处理

```
try:
    risky_operation()
except (TypeError, ValueError) as e:
    print(f"捕捉到异常: {e}")
except Exception:
    print("其他异常")
else:
    print("无异常时执行")
finally:
    print("无论是否异常都执行")
```

- else 只在 try 无异常时执行（用 else 而非把代码放在 try 里，能精确表达意图）。

- finally 即使有 return/异常也会执行。而且会先执行
- **异常层级**：BaseException → Exception → 具体异常。KeyboardInterrupt 等继承自 BaseException 而非 Exception。
- **自定义异常**：继承 Exception，通常只定义类名，不做其他。
- **finally 块不会比 try 或 except 先执行。**
- **它是在 try 块（以及可能被触发的 except 块）执行完毕后，但在控制权真正离开 try...except...finally 结构（比如函数返回或异常抛出）之前执行的。**
- **出现异常，返回 None**

5. 上下文管理器（with 协议）

资源（文件、锁、网络连接）在不再需要时必须释放，否则会泄露。

```
with open('file.txt') as f:
    data = f.read()
```

原理：执行 __enter__ 返回对象，退出时执行 __exit__ 保证释放资源。

实现方式：

- 类实现 __enter__ 和 __exit__
- 使用 contextlib.contextmanager 装饰器定义生成器

```
from contextlib import contextmanager

@contextmanager
def file_manager(name, mode):
    f = open(name, mode)
    try:
        yield f
    finally:
        f.close()

with file_manager('test.txt', 'w') as f:
    f.write('hello')
```

yield **之前**的代码（如 open）	进入 with 块时立即执行	__enter__
yield **之后**的代码（如 f.close()）	with 块退出时执行（即使块内发生异常也会执行）	__exit__ 中的清理逻辑

- with 块执行用户代码，如果发生异常，Python 会调用上下文管理器的 __exit__ 方法。
- 对于 @contextmanager 装饰器生成的上下文管理器，它的 __exit__ 方法会将这个异常**注入**到生成器中，方式是通过生成器的 throw() 方法。

throw() 会在生成器上次暂停的位置（即 yield f 这一行）**抛出**该异常

所以，yield f 这一行就像被替换成了 raise 那个异常。生成器函数从暂停点恢复，并且立即遇到一个异常。

__exit__ 的三个参数：exc_type, exc_val, exc_tb，返回 True 可以压制异常。

- **当 with 块正常执行（无异常）**：三个参数都传入 None。
- **当 with 块内抛出异常**：
  - exc_type：异常的类型（例如 ValueError 类对象）。
  - exc_val：异常实例（通常与 exc_type 对应，如 ValueError("something")）。
  - exc_tb：异常的 traceback 对象（用于回溯栈帧）。
  - traceback 对象包含的信息**远超**仅仅是出错代码行。它记录了**从异常发生点向上追溯到调用入口的完整调用栈**。

__exit__ 方法可以返回一个布尔值：

- 返回 True 表示**吞没异常**（异常不会再向外传播）。
- 返回 False 或 None（默认）表示**不处理**，异常会继续向上抛出。

任何实现了__enter__ 和 __exit__方法的对象都可以用 with：

```
class ManagedFile:
    def __init__(self, name, mode):
        self.name, self.mode = name, mode
    def __enter__(self):
        self.file = open(self.name, self.mode)
        return self.file                    # 赋值给 as 后的变量
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.file.close()                   # 无论是否异常都执行
        # 返回 True 可以压制异常，但不建议
```

6. 常用模块速览（collections, itertools）

这部分你可以快速浏览，面试用到时能说出名字和用途即可。

- collections.namedtuple：有名字的元组，可读性好。
- collections.deque：双端队列，两端操作 O(1)。
- collections.Counter：统计可哈希对象的个数。
- collections.defaultdict：提供默认值的字典。
- itertools.chain：把多个迭代器串成一个。
- itertools.product：笛卡尔积。
- itertools.groupby：按相邻相同键分组。

一、对象模型（必考）

Q1：可变/不可变类型各举 5 个，核心区别是什么？

- 不可变：int float str tuple frozenset bytes（值不可改，改就是新对象）
- 可变：list dict set bytearray（可原地修改，id 不变）
- 核心：操作后 id() 变没变

Q2: == 和 is 的区别？

- == 比值（调 __eq__），is 比身份（比 id）
- 口诀：== 看长得像不像，is 看是不是同一个人

Q3: a=256; b=256 和 a=257; b=257，a is b 分别输出什么？

- 256 → True（小整数驻留 -5~256）
- 257 → False（超出范围，两个对象）
- 注意：这是 CPython 实现细节，不是语言规范

Q4：两个空列表 [] is [] 输出什么？

- False，每次 [] 都创建新列表对象

二、函数传参与参数（必考）

Q5: Python 传参是传值还是传引用？

- 传对象引用（pass by assignment）
- 口诀：形参实参贴同一个对象；改内容影响外部，改指向不影响外部

Q6：默认参数有什么陷阱？如何避免？

- 默认参数在函数定义时只计算一次，可变默认值会累积
- 修正：用 None 做哨兵，函数内判空再初始化

```
def f(lst=None):
    if lst is None:
        lst = []
```

Q7：参数顺序是什么？

- def f(pos, *args, kw=默认, **kwargs)
- 调用顺序：位置 → 可变位置 → 关键字 → 可变关键字

三、作用域与闭包（高频）

Q8: LEGB 是什么？

- Local → Enclosing → Global → Built-in
- 变量查找沿这个链条从内向外

Q9: global 和 nonlocal 的区别？

- global：在函数内声明使用全局变量
- nonlocal：在内层函数声明使用外层函数的变量（闭包场景）

Q10：闭包是什么？举例说明。

- 内层函数引用了外层函数的变量，并且外层函数已执行完，内层函数还能记住那个变量

```
def outer(x):
    def inner(y):
        return x + y
    return inner
```

Q11：这个代码输出什么？

python

```
funcs = [lambda: i for i in range(3)]
print([f() for f in funcs])
```

- [2,2,2]，lambda 延迟绑定，取到循环结束后的 i
- 修正：lambda i=i: i 用默认参数立即绑定

四、装饰器（必问）

Q12: 装饰器本质是什么？写一个计时装饰器。

- 本质：接受函数返回新函数的可调用对象
- 语法糖：@decorator 等价于 func = decorator(func)

```
from functools import wraps
import time

def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"耗时: {time.time()-start:.4f}s")
        return result
    return wrapper
```

Q13: @wraps 是干什么的？

- 保留原函数的 __name__、__doc__ 等元信息，不然会变成 wrapper 的信息

Q14: 带参数装饰器的结构？

- 三层嵌套：外层收参数 → 中层真正装饰器 → 内层 wrapper

```
def retry(times=3):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            ...
        return wrapper
    return decorator
```

- 原理：@retry(times=5) 先调 retry(5) 返回 decorator，再装饰

五、生成器（中频）

Q15: yield 和 return 的区别？

return	yield
函数结束，返回一个值	函数暂停，返回一个值，状态保留
调用完状态丢失	下次 next() 从暂停处继续

Q16: 生成器表达式和列表推导式的区别？

- 列表推导：[x**2 for x in range(n)] 立即生成全部，占内存
- 生成器表达式：(x**2 for x in range(n)) 惰性求值，省内存

Q17: 同一个生成器对象，list(gen) 执行两次，第二次会怎样？

- 第一次消耗完所有元素
- 第二次返回空列表 []

六、类与面向对象（高频）

Q18: 类变量和实例变量的区别？陷阱是什么？

- 类变量：所有实例共享，定义在类体内方法外
- 实例变量：各实例独有，在 __init__ 中用 self.xxx 定义
- 陷阱：可变类变量被实例通过 self.xxx.append() 修改会影响所有实例
- 修正：可变属性放 __init__ 里

Q19: self 是什么？

- 代表实例本身，Python 自动把调用者传进方法的第一个参数

Q20: super() 的作用？

- 按 MRO 顺序找下一个类的方法，不是直接调父类
- 在多继承中保证合作式调用

Q21: 菱形继承 MRO 顺序怎么看？

```
class D(B, C): pass   # B、C 都继承 A
print(D.__mro__)   # D → B → C → A → object
```

- 原则：子类在父类前，保持定义顺序，每个类只出现一次

Q22: @classmethod 和 @staticmethod 区别？

	@classmethod	@staticmethod
第一参数	cls（类）	无
访问类属性	可以	不能直接（需类名）
典型用途	工厂方法	工具函数

七、内存与拷贝（中频）

Q23: 浅拷贝和深拷贝的区别？

- 浅拷贝 copy.copy：只复制外层容器，内部元素共享引用
- 深拷贝 copy.deepcopy：递归复制整个对象树，完全独立
- 口诀：浅拷贝只深一层

Q24: Python 的垃圾回收机制？

- 引用计数为主，引用为 0 时立即回收
- 标记-清除解决循环引用
- 分代回收提高效率（0/1/2 代，新对象在 0 代，回收最频繁）

Q25: del a 一定会回收对象吗？

- 不一定，只是引用减 1
- 引用计数不为 0 不回收
- 循环引用需等标记-清除

八、异常处理（中频）

Q26: try/except/else/finally 的执行顺序？

- try 正常 → else → finally
- try 异常 → except → finally
- finally 无论如何都会执行

Q27: finally 里有 return 会怎样？

- 覆盖 try 或 else 中的返回值
- 同时压制未处理的异常（谨慎使用）

Q28: 自定义异常怎么写？

```
class MyError(Exception):
    pass
```

九、上下文管理器（中频）

Q29: with 语句的底层协议是什么？

- __enter__：进入时调用，返回值赋给 as 后的变量
- __exit__：退出时调用，三个参数 (exc_type, exc_val, exc_tb)

Q30: __exit__ 返回 True 会怎样?

- 压制 with 块内的异常，不向外传播

Q31: 用 contextlib.contextmanager 写一个上下文管理器。

```
from contextlib import contextmanager

@contextmanager
def timer():
    import time
    start = time.time()
    try:
        yield
    finally:
        print(f"耗时: {time.time()-start:.4f}s")
```

- yield 前：进入逻辑
- yield 后：退出逻辑（try...finally 保证异常时也执行）

十、内置函数（低中频）

Q32: map/filter/reduce 怎么用?

- map(fn, iter)：每个元素调用 fn，返回迭代器
- filter(fn, iter)：保留 fn 返回 True 的元素
- reduce(fn, iter)：累积计算，在 functools 里

Q33: 字典按值排序怎么写?

```
sorted(d.items(), key=lambda x: x[1], reverse=True)
```

附：面试策略速记

场景	应对
遇到不会的	"我目前了解不深，但按我的理解..."
被问代码量	"独立完成过 XX 项目，约 XXX 行，用了..."
被追问	先说结论，再展开，最后一句总结
要求手写代码	先确认输入输出，考虑边界，写完自测一个用例