Day 03 Python 期末复习笔记合集

按 day 目录合并生成

来源：C:\Users\50469\python-learning\期末复习笔记\day03

目录

1. theory_11_closure_decorator.md
2. theory_25_function_basics.md
3. theory_25_recursion.md
4. 笔记_内置函数速查.md
5. 笔记_高阶函数.md
6. 课堂老师程序-函数专题对照.md

1. theory_11_closure_decorator.md

- 11-closure-decorator

闭包 + 装饰器：搞懂 @xxx 到底在干什么

前置：已掌握函数定义、函数嵌套、`@property` 和 `@classmethod`
目标：理解装饰器原理，看懂 FastAPI 的 `@app.get("/path")`
用时：约 15 分钟

相关笔记：函数基础 - Python 基础补漏（作用域/*args） - 课堂老师函数程序对照

一、热身：函数也是变量

先接受一个事实：Python 里函数和数字、字符串一样，可以赋值、可以当参数传。

```
def say_hello(name):
    return f"■■, {name}!"

# ■■■■■■■■■■
greeter = say_hello
print(greeter("■■"))   # ■■, ■■!

# ■■■■■■■■■
def run(func, arg):
    return func(arg)

print(run(say_hello, "■■"))  # ■■, ■■!
```

这不神奇--只是 Python 把函数当作"一等公民"（first-class citizen），和整数、字符串地位一样。

二、闭包：函数记住了它出生时的环境

一个现象

```
def make_multiplier(n):
    def multiplier(x):
        return x * n       # multiplier ■■■■■ n
    return multiplier

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))   # 10
print(triple(5))   # 15
```

`make_multiplier(2)` 结束之后，按理说变量 `n=2` 应该消失了--但 `double(5)` 仍然记得 `n=2`。

这就是闭包：内部函数 `multiplier` 把外部函数 `make_multiplier` 的变量 `n` 给"记住"了。

闭包的三个条件

1. 函数里定义函数（嵌套）
2. 内部函数用了外部函数的变量
3. 外部函数把内部函数返回出去

闭包的经典用途：计数器

```
def create_counter(start=0):
    count = start
    def counter():
        nonlocal count       # <- ■■ count ■■■■■■■■■■■■■
        count += 1
        return count - 1
    return counter

c = create_counter(10)
print(c())   # 10
print(c())   # 11
print(c())   # 12
```

`nonlocal` 的作用：告诉 Python "这个 `count` 不是本函数的局部变量，来自外层函数"。没有 `nonlocal`，内部函数里给 `count += 1` 会报错。

```
# ■■ nonlocal ■■■
def bad_counter(start=0):
    count = start
    def counter():
        count += 1     # [X] UnboundLocalError
        return count
    return counter
```

为什么需要 `nonlocal`：Python 的变量赋值机制--`count += 1` 等价于 `count = count + 1`，这会在内部函数新建一个局部变量 `count`，和外层的 `count` 没关系。`nonlocal` 说"别新建，用外层的"。

一句话记

> 闭包 = 内部函数 + 它记住的外部变量。`nonlocal` 让内部函数能修改这些"记住的"变量。

三、装饰器：给函数套一层包装

装饰器是什么

装饰器 = 接收一个函数，返回一个新函数的函数。

```
def log_calls(func):
    """■■■■■■■■■■■■■"""
    def wrapper(*args, **kwargs):
        print(f"■■ {func.__name__}()")
        result = func(*args, **kwargs)
        print(f"{func.__name__}() ■■ {result}")
        return result
    return wrapper
```

用法：

```
# ■■1■■■■
def add(a, b):
    return a + b

add = log_calls(add)     # add ■■■■■ wrapper
print(add(3, 5))
# ■■ add()
# add() ■■ 8
# 8

# ■■2■@ ■■■■■■■■■■
@log_calls
def add(a, b):
    return a + b
```

```
# ↑ ■■■■■■■ add = log_calls(add)
```

`@` 只是一个快捷写法，它把下面的函数作为参数传给装饰器，然后用返回值替换原函数。

装饰器的数据流

```
@log_calls
def add(a, b):
    return a + b
```

等价于：

```
add = log_calls(add)
     ↓
log_calls ■■ add ■■
     ↓
■■ wrapper ■■
     ↓
add ■■■■ wrapper
     ↓
■■ add(3,5) -> ■■■■ wrapper(3,5)
     ↓
wrapper ■■■ -> ■■ add(3,5) -> ■■■■
```

你已经用过的装饰器

```
@property         # ■■■■■■■■■■
@classmethod    # ■■■■■■■■ cls ■■■ self
@staticmethod   # ■■ self ■ cls
```

为什么要用闭包来实现装饰器？

因为装饰器需要记住原函数（外部变量 `func`），并在 `wrapper` 里调用它--这天然就是个闭包。

```
def log_calls(func):       # <- ■■■■■■■■■■■
    def wrapper(*args, **kwargs):   # <- ■■■■■■■■■
        # func ■ wrapper ■■■■■■■■
        result = func(*args, **kwargs)
        return result
    return wrapper          # <- ■■■■■■■
```

☐ 破坏实验

```
# ■■■■■■■■■■■■■
def do_nothing(func):
    return func          # ■■■■■■■■

@do_nothing
def hello():
    return "Hello!"

print(hello())    # ■■■■■■■■■

# ■■■■■■■■■■■■■■■
def return_int(func):
    return 42

@return_int
def hello():
    return "Hello!"

print(hello)      # ■hello ■■■■■■■
# hello()         # ■■■■■■■■
```

四、带参数的装饰器：@app.get("/path")

三层嵌套

FastAPI 里这种写法你见过：

```
app = FastAPI()

@app.get("/hello")
def hello():
    return {"msg": "Hello!"}
```

这里 `@app.get("/hello")` 不是普通的装饰器--它先调用了 `app.get("/hello")` 方法，这个方法的返回值才是真正的装饰器。

三层结构：

```
■■■■■■■■■■"/hello"■
   ↓
■■■■■■■■■■hello■
   ↓
■■■■■■■■■■wrapper■
```

模拟实现

```
class FakeApp:
    def __init__(self):
        self.routes = {}              # ■■■■

    def get(self, path):              # <- ■■■■■■■■■■■
        def decorator(func):          # <- ■■■■■■■■■■■■■■■■■
            self.routes[path] = func  #   -> ■■■■■■■■■■■■■
            return func               #   -> ■■■■■■■■■■■■■■■■■■■
        return decorator              # <- ■■■■■
```

执行流程：

```
@app.get("/hello")
def hello():
    return "Hello!"
```

等价于：

```
@app.get("/hello")   ->  app.get("/hello")  -> ■■ decorator
@decorator           ->  decorator(hello)   -> ■ hello ■■■ self.routes["/hello"]
def hello():
    return "Hello!"
```

完整的例子

```
app = FakeApp()

@app.get("/hello")
def hello():
    return "Hello!"

@app.get("/status")
def status():
    return {"status": "ok"}

print(app.routes)
# {'/hello': <function hello>, '/status': <function status>}
```

对比普通装饰器：

	普通装饰器	`@app.get("/path")`
层数	2层（外=收函数，内=包装）	3层（外=收参数，中=收函数，内=包装）
典型用途	打日志、计时、权限校验	路由注册、配置绑定
调用方式	`@decorator`	`@obj.method(arg)`

□ 破坏实验

```
# ■■■■■■■■■■■■
def bad_decorator(path):
    # ■■■■■■■■
    pass  # ■■■■■■■■■■

@bad_decorator("/test")   # ■■■■■■
def test():
    return "test"
```

五、多个装饰器叠加

```
def bold(func):
    def wrapper(*args, **kwargs):
        return f"<b>{func(*args, **kwargs)}</b>"
    return wrapper

def italic(func):
    def wrapper(*args, **kwargs):
        return f"<i>{func(*args, **kwargs)}</i>"
    return wrapper

@bold
@italic
def greet(name):
    return f"Hello, {name}!"

print(greet("World"))
```

输出什么？

执行顺序是从下往上装饰，从上往下执行：

```
■■ greet = "Hello, World!"

@italic ■■■ -> "<i>Hello, World!</i>"
    ↑ ■■■ italic

@bold ■■■ -> "<b><i>Hello, World!</i></b>"
    ↑ ■■■ bold
```

等价于：

```
greet = bold(italic(greet))
```

> 装饰器叠加顺序：离函数最近的最先装饰，离函数最远的最后包装。

总结

概念	一句话	什么时候用
函数即变量	函数可以赋值、当参数传	理解装饰器的基础

概念	一句话	什么时候用
闭包	内部函数记住外部变量	计数器、缓存、装饰器底层实现
装饰器（2层）	`@xxx` = 接收函数，返回新函数	打日志、计时、权限校验
装饰器（3层）	`@xxx(arg)` = 先调外层收参数，返回装饰器	FastAPI 路由注册、配置绑定
叠加	`@A @B` = `A(B(func))`	多个独立关注点组合

2. theory_25_function_basics.md

- 25-function-basics

函数基础：定义 / 参数 / 返回值

前置：已掌握变量和条件语句	
目标：能独立写出带参数和返回值的函数	
用时：约 12 分钟	

相关笔记：递归函数 - 闭包与装饰器 - 作用域规则 - *args - **kwargs - 课堂老师函数程序对照

一、为什么要用函数

```
# ■■■■■■■■■■
print("■■■■■■■")
# ... 50 ■■■ ...
print("■■■■■■■")
# ... ■■■50■■■ ...
print("■■■■■■■")
# ... ■■■50■■■ ...

# ■■■■■■■■■■■■■
def welcome(name):
    print(f"■■{name}■■")
    # ... 50 ■■■ ...

welcome("■■")
welcome("■■")
welcome("■■")
```

函数的好处：消除重复、逻辑复用、方便修改。

二、定义和调用

```
# ■■
def greet():
    print("Hello!")

# ■■
greet()   # Hello!
```

结构：

```
def ■■■(■■■■):
    """■■■■■■■■■■"""
    ■■■
    return ■■■■■■■■
```

规则：

- `def` 关键字开头
- 函数名 + 圆括号 + 冒号
- 函数体缩进
- 参数和 `return` 都是可选的

三、参数

位置参数（最常用）

```
def greet(name, greeting):
    print(f"{greeting}, {name}")

greet("■■", "■■")   # ■■, ■■
greet("■■", "Hello")  # Hello, ■■
```

参数的顺序很重要，按位置一一对应。

关键字参数

```
# ■■■■■■■■■■■■■■
greet(greeting="■■", name="■■")   # ■■, ■■
```

默认参数（缺省参数）

```
def greet(name, greeting="■■"):
    print(f"{greeting}, {name}")

greet("■■")            # ■■, ■■ - ■■■■■■
greet("■■", "Hello")   # Hello, ■■ - ■■■■■■
```

默认参数陷阱：默认值只在定义时计算一次！

```
def add_item(item, lst=[]):   # [X] ■■
    lst.append(item)
    return lst

print(add_item(1))   # [1]
print(add_item(2))   # [1, 2] - ■■■■■■ [2]■

# [OK] ■■■■■■■ None ■■■■■
def add_item(item, lst=None):
    if lst is None:
        lst = []
    lst.append(item)
    return lst
```

四、返回值

```
# ■■■■■■
def add(a, b):
    return a + b

result = add(3, 5)
print(result)   # 8

# ■■■■■■■■■■■■■■■■
def get_min_max(nums):
    return min(nums), max(nums)

result = get_min_max([3, 1, 4, 1, 5])
print(result)         # (1, 5)
print(type(result))   # <class 'tuple'>

# ■■■■■
min_val, max_val = get_min_max([3, 1, 4, 1, 5])
print(f"■■■: {min_val}, ■■■: {max_val}")

# ■■ return -> ■■ None
def say_hello(name):
```

```
    print(f"■■■{name}")

result = say_hello("■■")   # ■■■■■
print(result)               # None
```

五、文档字符串（docstring）

```
def calculate_bmi(weight, height):
    """■■BMI■■

    ■■:
        weight: ■■■■■■
        height: ■■■■■

    ■■:
        BMI ■■float■
    """
    return weight / (height ** 2)

# ■■■■
help(calculate_bmi)   # ■■■■■ docstring
print(calculate_bmi.__doc__)  # ■■■■■■■■
```

团队协作必写 docstring，考试和面试也会加分。

六、类型提示（Type Hints）

1. 基础语法

```
# ■■■■■■■■: ■■ = ■
name: str = "■■"
age: int = 20

# ■■■■■■■■■■: ■■
# ■■■■■■■-> ■■
def greet(name: str, age: int) -> str:
    return f"{name} ■■ {age} ■"

print(greet("■■", 20))   # ■■ ■■ 20 ■
```

写法：变量名在前，冒号，类型。■■■: ■■ 读作"这个变量的类型是"。

```
price: float = 99.8          # ■■■
is_active: bool = True       # ■■■
tags: list[str] = ["a", "b"] # ■■■■■
```

2. 联合类型：一个字段可能是多种类型

```
# str | None  = "■■■■■■■■■■■ None■■■"
content: str | None = None

# ■■■■■■■
from typing import Optional
content: Optional[str] = None
```

"为什么类型写在变量名后面？"

> Python 语法设计就是 ■■■: ■■，不是 ■■ ■■■（后者是 Java/C 的写法）。
>
> Java: `String name = "■■"`
>
> Python: `name: str = "■■"`

读法一样："name 这个变量，类型是 str，值是'张三'"。

3. 为什么类型注解很重要

```
# ■■■■■■■■■■■■■■■■■■■■■■
def process(data, config):
    pass

# ■■■■■■■■■■■■■■■
def process(data: list[int], config: dict[str, str]) -> bool:
    pass
```

类型注解的好处：

1. IDE 自动补全（输入 `data.` 会弹出列表方法）
2. 代码自文档化（不用看函数体就知道参数要求）
3. Pydantic 用类型做校验（`content: str` -> 自动校验传进来的是不是字符串）
4. mypy 静态检查（在运行前发现类型不匹配）

4. 常见类型写法速查

```
# ■■■■
name: str = "abc"
count: int = 42
ratio: float = 3.14
done: bool = True

# ■■■■■Python 3.9+ ■■■■■■■■■■ typing ■■■
items: list[int] = [1, 2, 3]
mapping: dict[str, int] = {"a": 1}
unique: set[str] = {"x", "y"}
pair: tuple[str, int] = ("abc", 42)

# ■■■■■Python 3.10+■
maybe: str | None = None
value: int | str = 42  # ■■■■■■■■■■■■■■

# ■ Pydantic ■■■■■■
class NoteCreate(BaseModel):
    content: str                    # ■■■■■■■■■■

class NoteUpdate(BaseModel):
    content: str | None = None      # ■■■■■■■■■■■■
```

5. 重要提醒

类型注解不会影响运行--即使你标注 `name: str` 然后赋值数字，Python 也不会报错。它是给人和工具看的约定。

七、函数是第一公民

```
# ■■■■■■■■■■
def square(x):
    return x ** 2

f = square          # f ■■■ square ■■■■■
print(f(5))         # 25

# ■■■■■■■■■■■
def apply(func, value):
    return func(value)
```

```
result = apply(square, 4)
print(result)       # 16
```

总结

概念	写法	说明
定义	`def ■■■():`	def + 函数名 + 冒号
参数	`def f(x, y):`	位置参数按顺序传
关键字参数	`f(y=1, x=2)`	调用时指名
默认参数	`def f(x=10):`	参数有默认值
返回值	`return ■`	可返回多个（元组）
无返回值	不写 return	返回 None
文档	`"""■■"""`	函数体第一行
类型提示	`def f(x: int) -> str:`	代码更清晰

备考相关

- EXAM_PREP/day01/00_今日任务 - Day 1 基础语法（预热）
- EXAM_PREP/day03/00_今日任务 - Day 3 函数专项
- EXAM_PREP/day06/00_今日任务 - Day 6 老师课堂代码重放
- EXAM_PREP/day07/00_今日任务 - Day 7 学生管理系统综合题

3. theory_25_recursion.md

- recursion
- 递归函数

递归函数：函数自己调用自己

前置：已掌握函数定义、参数、返回值、if 判断
目标：能写出带终止条件的递归函数，理解实验题 `f(n)` 求 `1~n` 之和
用时：约 10 分钟

相关笔记：函数基础 - 条件判断 - 循环深入

一、递归是什么

递归就是：一个函数在函数体内部调用自己。

```
def count_down(n):
    if n == 0:
        print("■■")
        return

    print(n)
    count_down(n - 1)


count_down(3)
```

执行过程：

```
count_down(3)
  print(3)
  count_down(2)
    print(2)
    count_down(1)
      print(1)
      count_down(0)
        print("■■")
```

记忆句：递归 = 把大问题拆成一个小一点的同类问题。

二、递归必须有两个部分

1. 终止条件

告诉函数：什么时候别再调用自己了。

```
if n == 1:
    return 1
```

没有终止条件，函数会一直调用自己，最后报错：

```
RecursionError: maximum recursion depth exceeded
```

2. 递归关系

告诉函数：当前问题如何变成更小的问题。

```
return n + f(n - 1)
```

这句话的意思是：

```
1~n ■■ = n + 1~(n-1) ■■
```

三、实验题：递归求 1~n 之和

题目：编写函数 `f(n)`，输入 `n`，求 `1~n` 之和，要求使用递归。

```
def f(n):
    if n == 1:
        return 1

    return n + f(n - 1)


num = int(input("■■■ n■"))
print(f"1~{num} ■■■■{f(num)}")
```

以 `f(5)` 为例：

```
f(5)
= 5 + f(4)
= 5 + 4 + f(3)
= 5 + 4 + 3 + f(2)
= 5 + 4 + 3 + 2 + f(1)
= 5 + 4 + 3 + 2 + 1
= 15
```

关键点：每次调用都让 `n` 变小，直到 `n == 1`。

四、递归和循环的对比

递归写法：

```
def f(n):
    if n == 1:
        return 1
    return n + f(n - 1)
```

循环写法：

```
def f_loop(n):
    total = 0

    for i in range(1, n + 1):
        total += i

    return total
```

对比：

写法	优点	缺点
递归	思路像数学公式，适合树、分治、阶乘	层数太深会报错，初学容易忘终止条件
循环	更直观、更省内存	有些复杂结构写起来不如递归自然

考试里如果要求"使用递归"，就不能用 `for` 或 `while` 直接累加。

五、经典例子：阶乘

阶乘公式：

```
n! = n * (n-1) * (n-2) * ... * 1
```

递归关系：

```
n! = n * (n-1)!
```

代码：

```
def factorial(n):
    if n == 1:
        return 1

    return n * factorial(n - 1)


print(factorial(5))  # 120
```

展开过程：

```
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120
```

六、递归常见错误

错误 1：没有终止条件

```
def f(n):
    return n + f(n - 1)
```

问题：函数永远不知道什么时候停。

错误 2：参数没有变小

```
def f(n):
    if n == 1:
        return 1
    return n + f(n)
```

问题：每次还是 `f(n)`，没有靠近终止条件。

错误 3：终止条件写错

```
def f(n):
    if n == 0:
        return 0
    return n + f(n - 1)
```

这段代码本身可以算 `1~n`，但如果题目要求 `n` 是正整数，通常更自然写成：

```
if n == 1:
    return 1
```

如果要兼容 `n == 0`，可以写：

```
def f(n):
    if n <= 0:
        return 0
    return n + f(n - 1)
```

七、怎么判断一道题能不能用递归

看它能不能拆成：

```
■■■■ = ■■■■ + ■■■■■■■■
```

适合递归的题：

- `1~n` 求和：`f(n) = n + f(n-1)`
- 阶乘：`factorial(n) = n * factorial(n-1)`
- 倒计时：`count_down(n)` 调 `count_down(n-1)`
- 树结构、目录遍历、分治算法

不太适合初学者用递归硬写的题：

- 简单列表遍历
- 普通菜单循环
- 学生管理系统这种交互式程序

八、考试速记

递归函数三步：

```
1. ■■■■■■■■
2. ■■■■■■
3. ■■■■■■■■■■■■■■■■■
```

模板：

```
def ■■■(n):
    if ■■■■:
        return ■■■■■■■■■

    return ■■■■ + ■■■(■■■■■■)
```

实验 7 的递归求和模板：

```
def f(n):
    if n == 1:
        return 1
    return n + f(n - 1)
```

一句话复述：

> 递归就是函数自己调用自己，但必须有终止条件，并且每次调用都要让问题变小。

4. 笔记_内置函数速查.md

Python 常用内置函数速查

> 内置函数 = 不需要 `import`，直接就能用的函数

一、输入输出

```
print("hello")           # ■■■■■■■
print(1, 2, 3, sep="-")  # 1-2-3■sep ■■■■■■
print("a", end="")       # ■■■■■■■ end="\n"

input("■■■■■")          # ■■■■■■■■■■■■■■
name = input("■■■■")     # ■■■■ str ■■■■■■■■■■■■
```

二、类型转换

```
int("42")       # ■■■■ -> ■■■■42
int(3.9)        # ■■■■ -> ■■■■3■■■■■■■■■■■■
float("3.14")   # ■■■■ -> ■■■■■3.14
str(100)        # ■■■■■ -> ■■■■■"100"
bool(1)         # -> True■bool(0) -> False■bool("") -> False
list("abc")     # -> ['a', 'b', 'c']
tuple([1,2,3])  # -> (1, 2, 3)
set([1,2,2,3])  # -> {1, 2, 3}■■■■■
dict([("a",1)]) # -> {'a': 1}
```

三、序列操作

```
len([1, 2, 3])        # ■■■■3
len("■■■")            # ■■■■■■■2■■■■■■■■■■■■

range(5)              # -> 0,1,2,3,4
range(2, 5)           # -> 2,3,4
range(0, 10, 2)       # -> 0,2,4,6,8

enumerate(["a","b"])  # -> (0,"a"), (1,"b")■■■■■■■■
zip([1,2], ["a","b"]) # -> (1,"a"), (2,"b")■■■■■■■■■■

reversed([1,2,3])     # -> 3,2,1■■■■■■■■
sorted([3,1,2])       # -> [1,2,3]■■■■■■■■
sorted([3,1,2], reverse=True)  # -> [3,2,1]
```

四、数学相关

```
abs(-5)          # ■■■■■5
max(1, 3, 2)     # ■■■■■3
max([1, 3, 2])   # ■■■■■■
min(1, 3, 2)     # ■■■■■1
sum([1, 2, 3])   # ■■■■6
round(3.14159)   # ■■■■■■3
round(3.14159, 2) # ■■■2■■■■■3.14

pow(2, 3)        # 2³■8■■■■■ 2**3
divmod(10, 3)    # (■, ■■)■(3, 1)
```

五、类型判断与检查

```
type(123)        # <class 'int'>
type("abc")      # <class 'str'>
type([1,2])      # <class 'list'>

isinstance(123, int)        # True
isinstance("abc", str)      # True
isinstance([1,2], (list, tuple))  # ■■■■■■

id(123)          # ■■■■■■■■■■■■
id("hello")      # ■■■■■■■■■
```

六、其他常用

```
# ■■■■■■■■■■■■■
dir([])          # ■■■■■■■■■■

# ■■■■■■
all([True, 1, "a"])    # True■■■■■■■■■
any([0, "", "a"])      # True■■■■■■■■■

# ■■
help(print)      # ■■■■■■■■
```

七、一句话记

分类	记住这几个就行
输入输出	`print()` `input()`
类型转换	`int()` `str()` `float()` `bool()` `list()` `dict()`
序列	`len()` `range()` `enumerate()` `zip()` `sorted()` `reversed()`
数学	`max()` `min()` `sum()` `abs()` `round()`
检查	`type()` `isinstance()` `id()`

5. 笔记_高阶函数.md

高阶函数：函数作为参数

theory_11_closure_decorator - 课堂老师程序-函数专题对照

核心思想：把"做什么"和"什么时候做"分开

```
# ■■■■■■■■■--■■■■■■■■■■■■■
def run(func, *args, **kwargs):
    print("■■■■■")
    result = func(*args, **kwargs)
    print("■■■■■")
    return result
```

`run` 的价值不在于它能调用函数，而在于它能在调用函数前后做通用的事情。

一、对比：不用高阶函数 vs 用高阶函数

[X] 不用高阶函数：每个函数都要改

```
def say_hello(name):
    print("■■■■■")              # ■■■■■
    result = f"■■, {name}!"
    print("■■■■■")              # ■■■■■
    return result

def add(a, b):
    print("■■■■■")              # ■■■■■
    result = a + b
    print("■■■■■")              # ■■■■■
    return result

def multiply(a, b):
    print("■■■■■")              # ■■■■■
    result = a * b
    print("■■■■■")              # ■■■■■
    return result
```

问题： 100 个函数就要改 100 遍。

[OK] 用高阶函数：通用逻辑只写一次

```
def run(func, *args, **kwargs):
    print("■■■■■")
    result = func(*args, **kwargs)
    print("■■■■■")
    return result

# ■■■■■■■■
def say_hello(name):
    return f"■■, {name}!"

def add(a, b):
    return a + b

# ■■■■■■■■
run(say_hello, "■■")
run(add, 3, 4)
```

二、和装饰器的关系

```
# run ■■■■■■■■
def run(func, *args, **kwargs):
    print("■■■■")
    result = func(*args, **kwargs)
    print("■■■■")
    return result

# ■■■■■■■■■■--■■■■
def logger(func):
    def wrapper(*args, **kwargs):
        print("■■■■")
        result = func(*args, **kwargs)
        print("■■■■")
        return result
    return wrapper
```

区别： `run` 是手动传函数调用，装饰器是 `@logger` 标记后自动包装。

三、三个实战价值

1. 动态选择执行哪个函数（替代 if-elif 链）

```
def add(a, b): return a + b
def sub(a, b): return a - b
def mul(a, b): return a * b

# ■■■■■ if-elif
ops = {"+": add, "-": sub, "*": mul}

op = input("■■■: ")
a, b = int(input()), int(input())

if op in ops:
    print(ops[op](a, b))    # ■■■■■■
```

2. 批量执行多个函数

```
def square(x): return x ** 2
def cube(x): return x ** 3
def double(x): return x * 2

funcs = [square, cube, double]
for f in funcs:
    print(f(5))  # 25, 125, 10
```

3. 在调用前后做通用操作（日志/计时/权限）

```
import time

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

四、什么时候用

场景	直接调用	函数作为参数
只调一个特定函数	[OK] 推荐	[X] 多余
需要在调用前后做通用操作	[X] 重复代码	[OK] 一次编写到处用
需要动态选择执行哪个函数	[X] if-elif 冗长	[OK] 映射表
需要批量执行多个函数	[X] 一个个写	[OK] 简洁

五、传入函数和返回函数是对称的

```
■■■■■■■■■   ->   ■■■■■■■■■■■■■■■
                        ↕
■■■■■■■■■■ ->   ■■■■■■■■■■■■■■■/■■■
```

两者合在一起就是 Python 装饰器的完整实现。

6. 课堂老师程序-函数专题对照.md

date: 2026-05-27

source: C:\Users\50469\Desktop\python实验\课堂老师程序

status: 已整理

课堂老师程序对照：第六章函数

关联笔记： 25-function-basics/theory_25_function_basics|25-function-basics - 11-closure-decorator/theory_11_closure_decorator|11-closure-decorator - 16-python-gaps/theory_16_python_gaps|16-python-gaps - 10-fastapi/theory_10_fastapi|10-fastapi - 16-python-gaps/笔记_高阶函数

如果你只有时间读三句话

1. 老师这批代码在讲什么：Python 函数的核心机制--定义、传参、返回值、作用域、嵌套、装饰器。这是你当初看 FastAPI 一头雾水的底层原因。你的 25-function-basics/theory_25_function_basics|25-function-basics 是同一批知识的更详细版本。
2. 这和你什么关系：FastAPI 的 `@app.get` 本质就是装饰器（11-closure-decorator/theory_11_closure_decorator|11-closure-decorator 已经讲过），`def create_note(note: NoteCreate)` 本质就是函数定义，`Depends()` 本质就是框架帮你传参。不懂函数机制，FastAPI（10-fastapi/theory_10_fastapi|10-fastapi）就是空中楼阁。
3. 怎么用这份笔记：按顺序看，每个概念分四步--看老师代码 -> 看解释 -> 看 FastAPI 映射 -> 关文件自己说一遍。

一、函数定义和调用

老师代码

```
def fn():
    pass

def add():
    result = 11 + 22
    print(result)

def add_modify(a, b):
    result = a + b
    return result

# ■■■■■■■■■■■
x = add
x()

# ■■■■■■■
print(add_modify(10, 20))
```

它在说什么

现象	含义
`def fn(): pass`	定义一个什么都不做的函数
`add()` 输出 33	函数定义后不会自动执行，必须用 `()` 调用
`x = add; x()`	函数是第一类对象--可以赋值给变量
`add_modify(10, 20)`	带参数的函数：调用时传入实际值

理解：
函数是"先定义，后使用"的代码块。定义时只是注册，调用时才执行。这个区分是所有后续概念的基础。详细展开见25-function-basics/theory_25_function_basics|25-function-basics。

FastAPI 映射

对应 10-fastapi/theory_10_fastapi|10-fastapi
的路由注册、10-fastapi/theory_10_fastapi#16-Depends：让-FastAPI-帮你拿对象|Depends 依赖注入。

```
# ■■■■■■■■■■■■■
@app.post("/notes")
def create_note(note: NoteCreate, service: NoteServiceDep) -> NoteRead:
    return service.create_note(note)

# ■■■■■■■■--FastAPI ■■■■■
# ■■■■■ POST /notes -> FastAPI ■■■■■ -> ■■■ create_note(...)
```

☒ 为什么你不明白--因为你没见过"有人替你调用"这种模式

到目前为止你见过的所有函数调用都是 你主动写代码去调：

```python
result = add_modify(10, 20) # <- 你亲自调用的
```

但 FastAPI 的模式是：你只管定义函数，框架在合适的时机替你调用它。

对比	老师代码	FastAPI
调用者	你自己写 `add_modify(10, 20)`	FastAPI 框架自动调用
触发时机	代码执行到这行	收到 HTTP 请求时
参数来源	你手动传入	框架从 HTTP 请求提取
return 结果去哪	回到你的代码里	框架接管，序列化后发回浏览器

打个比方：

场景	类比
普通函数调用	你打电话给外卖店 -> 你主动拨号 -> 对方接听 -> 你拿到结果
FastAPI 路由	你开了一家店装了电话 -> 你告诉店员怎么接 -> 电话响时店员自动接 -> 按你教的回答

> 函数本身完全没变--都是 `def ■■■(■■): ■■■■ return ■■`。变的只是调用者从你变成了框架。

二、return——函数把结果交到哪里

老师代码

```
# return ■■■"■■"■■■
def filter_sensitive_words(words):
    if "■■" in words:
        new_words = words.replace("■■", "*")
    return new_words

result = filter_sensitive_words("■■■■■■■■■■■■")
print(result)  # ■■■■■*■■■■
```

核心：`return` 不是打印，是把值传递给调用者。没有 `return`，函数返回 `None`。

多个返回值本质是元组

```
def move(x, y, step):
    nx = x + step
    ny = y - step
    return nx, ny

# ■■■■
a, b = move(100, 100, 60)  # a=160, b=40

# ■■■■■■■
result = move(100, 100, 60)
print(type(result))  # <class 'tuple'>
print(result)        # (160, 40)
```

关键理解：Python 没有"返回多个值"，`return nx, ny` 实际上是 `return (nx, ny)`--一个元组。

FastAPI 映射

```
@app.post("/notes")
def create_note(note: NoteCreate, service: NoteServiceDep) -> NoteRead:
    return service.create_note(note)
    # return ■■ -> FastAPI -> ■■■ JSON -> ■■■■■
```

老师代码里 `return` 交给 `print`，FastAPI 里 `return` 交给框架序列化。本质都是把值传到函数外面。

> ☒ `return` 之后发生了什么--FastAPI 内部帮你做了 3 件事

步骤	发生了什么	代码层面
第 1 步：拿到返回值	`service.create_note(note)` 返回一个 Python 对象，比如 `NoteRead(id=1, content="■■", ...)`	对象在内存中
第 2 步：序列化	FastAPI 调 Pydantic 的 `.model_dump()` 把对象转字典，再转成 JSON 字符串	`{"id":1,"content":"■■"}` -> `'{"id":1,...}'`
第 3 步：发回浏览器	包装成 HTTP 响应报文，Uvicorn 通过 TCP 网络发出去	`HTTP/1.1 200 OK` + JSON 体

对比一下就懂了：

场景	流程
老师代码	`return a + b` -> `print(result)` <- 结果到控制台就结束了
FastAPI	`return obj` -> FastAPI 序列化 -> HTTP 响应 -> 网络传输 -> 浏览器看到 JSON

区别不是 `return` 变了，而是 return 之后接的东西不同。

三、参数传递--怎么把数据送进函数

3.1 位置参数

```
def get_max(a, b):
    if a > b:
        print(a, "■■■■■")
    else:
        print(b, "■■■■■")

get_max(8, 5)  # 8 ■■■■■ -- a=8, b=5■■■■■■
```

3.2 关键字参数

```
def fn(a, b):
    print(a, b)

fn(b=5, a=10)  # 10 5 -- ■■■■■■■■■■■■
```

☒ 通俗版：位置参数是"按座位坐"，关键字参数是"喊名字"

方式	类比	代码
位置参数	老师喊"按学号入座"--1 号坐第 1 个位，顺序必须对	`fn(8, 5)`
关键字参数	"张三坐李四的位置也行，只要你喊名字"--报名字就能对上	`fn(b=5, a=10)`

场景	用位置参数	用关键字参数
参数很少，一看就懂	`add(1, 2)` [OK]	没必要
参数很多，怕搞混顺序	容易传错 [X]	`create_user(name="■■", age=20, city="■■")` [OK]
只传其中几个参数	必须全传 [X]	`fn(a=10)` 只传 a [OK]

一句话：位置参数省事但死板，关键字参数啰嗦但保险。

3.3 仅限位置参数（/）

```
def func(a, b, /, c):        # / ■■ a,b ■■■■■■
    print(a, b, c)

func(10, 20, 30)             # ■■
```

```
func(10, 20, c=30)           # ■■
# func(a=10, b=20, c=30)     # ■■■a,b ■■■■■■■■
```

3.4 默认参数

```
def fn(a, b, c=100):        # c ■■■■ 100
    print(a, b, c)

fn(10, 20)                   # 10 20 100
fn(10, 20, c=30)             # 10 20 30
```

FastAPI 映射

对应 10-fastapi/theory_10_fastapi#6-路径参数：从-URL-路径里拿值|路径参数 和 10-fastapi/theory_10_fastapi#7-查询参数：从问号后面拿值|查询参数 两节。

```
@app.get("/notes/{note_id}")           # note_id ■ URL ■■■■■■■■■■
def get_note(note_id: int, q: str = None):  # q ■■■■■■■■■■
    ...

@app.get("/items")
def list_items(skip: int = 0, limit: int = 10):  # ■■■■■■■■
    ...
```

本质： 框架从 HTTP 请求中提取数据，按参数名/位置传给函数。你定义的参数就是框架填的槽。

四、*args 和 **kwargs——参数打包

打包

```
# *args■■■■■■■■■■■■■■
def test(*args):
    print(type(args))  # <class 'tuple'>
    print(args)        # (61, 22, 13, 44, 55)

test(61, 22, 13, 44, 55)

# **kwargs■■■■■■■■■■■■■■■
def test(**kwargs):
    print(type(kwargs))  # <class 'dict'>
    print(kwargs)        # {'a': 11, 'b': 22, 'c': 33, 'd': 44, 'e': 55}

test(a=11, b=22, c=33, d=44, e=55)
```

解包

```
# *tuple■■■■■■■■■■■
def test(a, b, c, d, e):
    print(a, b, c, d, e)

nums = (11, 22, 33, 44, 55)
test(*nums)  # ■■■ test(11, 22, 33, 44, 55)


# **dict■■■■■■■■■■■■
nums = {"a": 11, "b": 22, "c": 33, "d": 44, "e": 55}
test(**nums)  # ■■■ test(a=11, b=22, c=33, d=44, e=55)
```

理解： 打包和解包是对称的--

位置	* 的作用	** 的作用
定义形参时	`*args`：多个位置参数 -> 元组	`**kwargs`：多个关键字参数 -> 字典
调用实参时	`*tuple`：元组 -> 多个位置参数	`**dict`：字典 -> 多个关键字参数

详细讲解见 16-python-gaps/theory_16_python_gaps#5-args-和-kwargs|*args/**kwargs 和 16-python-gaps/笔记_高阶函数。

混合传递顺序规则

```
def test(a, b, c=33, *args, **kwargs):
    print(a, b, c, args, kwargs)

test(1, 2)              # 1 2 33 () {}
test(1, 2, 3)           # 1 2 3 () {}
test(1, 2, 3, 4)        # 1 2 3 (4,) {}
test(1, 2, 3, 4, e=5)   # 1 2 3 (4,) {'e': 5}
```

顺序铁律： 普通参数 -> 默认参数 -> `*args` -> `**kwargs`

FastAPI 映射

```
def new_function(*args, **kwargs):
    result = old(*args, **kwargs)
    return result
```

装饰器里 `*args, kwargs`
起到透明转发**的作用--包装函数不知道原函数签名，但用打包再解包就能原样转发。这是 FastAPI 装饰器能包装各种不同参数路由函数的原因，见 16-python-gaps/笔记_高阶函数 的 `run` 函数模式。

五、作用域--变量在哪里有效

局部变量

```
def test_one():
    number = 10       # ■■■■■■■■■■■■
    print(number)

test_one()            # 10
# print(number)       # ■■■■■■■■■■■

# ■■■■■■■■■■■■■■■■■■
def test_one():
    number = 10
    print(number)     # 10

def test_two():
    number = 20
    print(number)     # 20
```

全局变量--只能读不能直接改

```
number = 10           # ■■■■

def test_one():
    print(number)     # ■■■10
    # number += 2     # ■■■■■■■■■--Python ■■■■■■■■■■■
```

```
test_one()
print(number)         # 10■■■■
```

global——声明要改全局变量

```
number = 10

def test_one():
    global number     # ■■■■■■■■■■■■■ number
    number += 1
    print(number)     # 11

test_one()
print(number)         # 11■■■■■■
```

nonlocal——改外层函数的变量

```
def test(a):
    number = 10
    def test_in():
        nonlocal number   # ■■■■■■■■■■■■■■ number■■■■■
        number += 20
        print(number, a)  # 30 15
    test_in()
    print(number)         # 30■■■■■■■■

test(15)
```

理解： 变量查找顺序--局部 -> 外层函数 -> 全局 -> 内置（LEGB 法则，见 16-python-gaps/theory_16_python_gaps#10-变量作用域legb-规则|LEGB 规则）。

☒ `nonlocal` 可以连续用吗？-- 可以，它会一级一级往上链

你问的是这种场景：三层嵌套，每层都用 `nonlocal`，能不能一路改到最外层？

```python
def outer():
x = 1
def middle():
nonlocal x # middle 说：x 是 outer 的 x
x += 10
def inner():
nonlocal x # inner 说：x 是 middle 的 x（而 middle 的 x 就是 outer 的 x）
x += 100
inner()
print(f"middle 里 x = {x}") # 111
middle()
print(f"outer 里 x = {x}") # 111

outer()
```

执行顺序	x 的值	谁改的
初始	1	`outer` 定义
`middle()` 中 `x += 10`	11	`middle` 的 `nonlocal x` 指向 `outer`
`inner()` 中 `x += 100`	111	`inner` 的 `nonlocal x` 指向 `middle`（而 `middle` 的 `x` 就是 `outer` 的 `x`）

> 机制： `nonlocal` 不是"跳到指定层"，而是"找最近的外层"。如果最近的外层本身也是 `nonlocal`，它就顺着链条往上走，最终都指向同一个变量。

> 不过实际代码很少写三层 `nonlocal` 连环套--两层（闭包/装饰器）已经够用了。

关键字	作用域	用于
无	局部 / 全局	大多数情况
`global`	在函数内修改全局变量	配置、计数器
`nonlocal`	在内层函数修改外层函数的变量	闭包、装饰器

> `global` 和 `nonlocal` 就是告诉 Python"跳过局部这个层级，直接去更外层找"。

六、嵌套函数和闭包

```
def add_modify(a, b):
    result = a + b
    print(result)

    def test():              # ■■■■■■■
        print("■■■■■■■")

    test()                   # ■■■■■■■

add_modify(10, 20)           # 30 \n ■■■■■■■
# ■■■■■■■ test()--■■■ add_modify ■■■■
```

关键理解： 内层函数相当于外层函数的"私有工具"，外部代码看不到它。这是信息隐藏的一种方式。这是 11-closure-decorator/theory_11_closure_decorator#二、闭包：函数记住了它出生时的环境|闭包 的基础。

FastAPI 联系： 装饰器的 `new_function` 就是嵌套在 `begin_end` 内部的，装饰器的本质就是：

1. 外层函数接收原函数
2. 内层函数包装原函数（加额外逻辑）
3. 返回内层函数替换原函数

七、装饰器--函数之上的函数

老师代码

```
import time

def begin_end(old):
```

```
    """[redacted]/[redacted]/[redacted]"""
    def new_function(*args, **kwargs):
        print('[redacted]~~~~')
        time_begin = time.time()

        result = old(*args, **kwargs)  # [redacted]

        time_end = time.time()
        print('[redacted]~~~~')
        ms = (time_end - time_begin) * 1000
        print(f"{old.__name__} [redacted] {ms:.2f} [redacted]")
        return result
    return new_function

# [redacted]
def sub(a, b):
    time.sleep(1)
    return a - b

f2 = begin_end(sub)
print(f2(10, 20))   # sub [redacted]

# [redacted]@[redacted]
@begin_end
def add(a, b):
    time.sleep(2)
    return a + b

result = add(3, 2)   # [redacted] add = begin_end(add)
print(result)
```

装饰器拆解

```
@begin_end
def add(a, b): ...

            ↓ [redacted]

add = begin_end(add)
```

`@begin_end` 不是"声明"，是在函数定义后立即执行了一个函数调用。

执行过程:

```
add = begin_end(add)
      -> begin_end [redacted] add [redacted]
      -> [redacted] new_function[redacted] old=add[redacted]
      -> [redacted] new_function
      -> add [redacted] new_function

[redacted] add(3, 2) [redacted]
      -> [redacted] new_function
      -> [redacted] "[redacted]"
      -> [redacted] add(3, 2)
      -> [redacted] "[redacted]"
      -> [redacted]
```

FastAPI 映射

```
@app.get("/notes")
def list_notes(): ...

# [redacted]
# 1. app.get("/notes") [redacted]
# 2. [redacted] list_notes
# 3. [redacted] list_notes [redacted]
# 4. list_notes [redacted]
```

区别只是老师写的 `begin_end` 接收函数，FastAPI 的 `app.get("/notes")` 先接收路径参数再返回装饰器。多了一层参数，本质一样。详细见 11-closure-decorator/theory_11_closure_decorator#四、带参数的装饰器：@app-get("/path")。

> 理解： 装饰器 = 高阶函数（参数是函数，返回值也是函数）。它是 Python 让函数作为第一类对象这个特性的集大成者。

八、递归函数

```
# ■ n!
# ■■■■■n == 1 -> ■■ 1
# ■■■■■n > 1 -> n * func(n-1)

def func(num):
    if num == 1:
        return 1
    else:
        return num * func(num - 1)

num = int(input("■■■■■■■■■"))
result = func(num)
print(f"{num}! = %d" % result)
```

递归两要素：

1. 边界条件： 什么时候停止（`n == 1`）
2. 递归公式： 问题缩小一圈（`n -> n-1`）

执行 `func(4)` 的展开过程：

```
func(4) -> 4 * func(3)
               -> 3 * func(2)
                     -> 2 * func(1)
                           -> 1
                     -> 2 * 1 = 2
               -> 3 * 2 = 6
         -> 4 * 6 = 24
```

九、匿名函数 lambda

```
# lambda ■■■■ : ■■■
temp = lambda a, b: a + b
print(type(temp))  # <class 'function'>
print(temp(10, 20))  # 30

# ■■■
def temp(a, b):
    return a + b
print(temp(10, 20))  # 30
```

lambda 限制： 只能写一行表达式，不能写语句。适合简单回调、排序 key、Pandas apply 等场景。你的 25-function-basics/theory_25_function_basics|25-function-basics 和 04-listcomp-class/theory_04_listcomp_lambda_class|04-listcomp-class 笔记中都用过它。

十、知识图谱--函数专题全景

```
■■ ■■■■ (def)
■   ■■ ■■■■■■■■■■■■■■■■
■■ ■■■ (return)
■   ■■ ■■■
```

```
                    ■   ■■ ■■■ -> ■■■■■
          ■■ ■■■■ ■■■
          ■            ■■ ■■■■
          ■            ■   ■■ ■■■■
          ■            ■   ■■ ■■■■■
          ■            ■   ■■ ■■■■■■■ (/)
          ■            ■   ■■ ■■■■
          ■            ■   ■■ ■■ *args / **kwargs
          ■            ■   ■■ ■■ *tuple / **dict
          ■            ■   ■■ ■■■■■■■■■
          ■            ■
          ■            ■■ ■■■
          ■                ■■ ■■■■
          ■                ■■ ■■■■■■■■■
          ■                ■■ global■■■■■■■■■
          ■                ■■ nonlocal■■■■■■■■■■■■
Python    ■
  ■■    ■■ ■■■■ ■■■
  ■■    ■            ■■ ■■■■■■■■■■■■■■■■■■
          ■            ■■ ■■■■■ + ■■■■■
          ■            ■■ ■■■■ lambda■■■■■■■■
          ■
          ■■ ■■■■ ■■■ ■■■
          ■            ■■ ■■■■■■■■■■■■->■■■■■■
          ■            ■■ @■■■■■@foo ≡ func = foo(func)
          ■            ■■ *args, **kwargs ■■■■
          ■            ■■ ■■■■■■■■■■■■■■■
          ■
          ■■ FastAPI ■■■
                       ■■ @app.get("/path") -> ■■■■■■■■
                       ■■ def create_note(...) -> ■■■■
                       ■■ Depends() -> ■■■■■■■
                       ■■ return -> ■■■■■■■■
```

十一、自检问题

学完之后，每道题能脱口而出才算真的懂了：

基础部分：

1. `def add(a, b): return a + b` 定义后，`add` 本身是什么类型？`add(1, 2)` 是什么？
2. `return` 和 `print` 的根本区别是什么？
3. 为什么 Python 可以 `a, b = func()` 接收两个返回值？底层发生了什么？
4. `def fn(a, b, /, c)` 中 `/` 的作用是什么？
5. `*args` 得到什么类型？`**kwargs` 得到什么类型？

进阶部分：

6. `test(*nums)` 是在做什么？`test(**nums)` 呢？用一句话说清楚。
7. 为什么函数内可以读全局变量，但不能直接改？
8. `global` 和 `nonlocal` 的区别一句话说清。
9. 装饰器 `@begin_end` 等价于哪一行普通代码？

FastAPI 连接部分：

10. `@app.get("/notes")` 和普通装饰器有什么区别和联系？
11. FastAPI 里 `return` 的值去了哪里，和老师代码有什么区别？
12. 为什么装饰器里要用 `*args, **kwargs` 来调用原函数？

13. FastAPI 的路径参数 (`/{id}`)、查询参数 (`?skip=0`)、请求体 (`POST body`) 分别对应函数参数的哪种传递方式?

附录：最快消化路线（40 分钟版）

按顺序跑，每个文件跑完说一句"这个文件教会了我什么"：

时间	文件	一句话目标
5 分钟	[redacted].py + [redacted].py	看懂函数定义->调用->return 的完整链路
5 分钟	[redacted].py	理解"多个返回值"本质是元组
5 分钟	[redacted]	三种传参方式
5 分钟	[redacted]1+2 + [redacted]1+2	掌握 `*`/`**` 在定义和调用时的对称用法
5 分钟	[redacted] + `nonlocal`	理解外层/内层函数关系
10 分钟	[redacted].py	最核心：理解装饰器本质 = 高阶函数
5 分钟	回头看 `notes-api` 的 `@app.post` 和 `def create_note`	验证你是否真懂了