

| STAT243 Lecture 5.6 Functional Programming

| 1 函数式编程

| 1.1 函数式编程概述

- **定义与目标**
 - 一种强调 **模块化、可复用、无副作用** 的编程范式；函数只依赖传入的参数，不使用全局变量，且尽量不产生 side effects。
 - 函数被视为 **first-class citizens**：可以作为参数传递、作为返回值、赋给变量使用。
- **匿名函数**
 - 在 Python 与 R 中可使用 **anonymous functions / lambda functions** 按需即时创建，用于临时的函数式操作场景。
- **在 Python 中进行 functional programming**
 - 通过编写 **自包含的函数** 而非类，利用函数的一等公民特性实现组合与复用。
 - **不完全符合的地方**：
 - Python 的 **pass-by-reference** 行为可能导致副作用：若在函数内修改 **mutable** 实参（如 list、numpy array），调用方对象会被更改；**immutable**（如 tuple）不受影响。
 - 一些操作以 **statements** 形式出现而不是函数调用（例如 `import`、`def`），与“万物皆函数”的理念不完全一致。
- **与 R 的对比**
 - R 函数更接近 **pass-by-value** 语义，更符合“纯”函数式风格；从范式一致性上更贴近 functional programming 的原则。

| 1.2 无副作用原则

- **核心思想**
 - 函数式编程追求“**无副作用 (no side effects)**”：函数只依赖输入参数并返回输出，而不改变程序外部状态（如全局变量或外部环境）。
 - 这样每个函数都可以被看作一个“**黑箱 (black box)**”，用户无需关心函数内部实现，也不必担心它会修改已有变量。
 - 程序运行结果可被视为数学意义上的函数复合（function composition）。

- **R 的实现方式**
 - R 中的大多数函数（以及理想的自定义函数）都遵循“输入 → 输出”的模式，不影响外部环境。
 - R 采用 **pass-by-value (按值传递)**：
 - 当对象被作为参数传入函数时，会在函数作用域中创建一个**局部副本**。
 - 在函数中修改该副本不会影响外部的原始对象。
 - 示例：

```
R
1 x <- 1:3
2 myfun <- function(x) {
3   x[2] <- 7
4   print(x)
5   return(x)
6 }
7 new_x <- myfun(x) # [1] 1 7 3
8 x               # [1] 1 2 3 (未被修改)
```

- 因此，R 的默认行为天然支持“无副作用”的原则。

- Python 的对比

- Python 使用 **pass-by-reference** (按引用传递):

- 当可变对象 (如 list、numpy array) 作为参数传入函数时, 函数中对该对象的修改会**直接影响原对象**。
 - 示例:

```
Python
1 x = np.array([1,2,3])
2 def myfun(x):
3     x[1] = 7
4     return x
5
6 new_x = myfun(x)
7 x # array([1, 7, 3]) ← 被修改
```

- 若函数不返回结果而直接修改对象 (如下例), 副作用更加明显:

```
Python
1 def myfun(x):
2     x[1] = 7
3     return None
4 myfun(x)
5 x # array([1, 7, 3])
```

- 因此, 在 Python 中**非常容易违反无副作用原则**。
若要避免, 应在函数内**显式复制对象**:

```
Python
1 def myfun(x):
2     y = x.copy()
3     y[1] = 7
4     return y
```

- 总结: R 与 Python 的差异

特性	R	Python
参数传递方式	Pass-by-value	Pass-by-reference
默认是否易产生副作用	否	是 (尤其是 mutable 对象)
一致性与可预测性	高	需谨慎设计函数以避免意外修改

- 例外情况

- 即使在 R 中, 也存在必要的副作用函数 (出于语言或交互性需求):
 - `par()`: 修改图形参数
 - `library()`: 加载包并改变环境状态
 - `plot()`: 绘制图形 (产生视觉输出)
 - 这些副作用是受控且必要的, 不影响函数式编程整体的可理解性。

1.3 函数是 first-class objects

- 在 Python 中, **一切皆对象 (Everything is an object)**
 - 在 Python 中, **函数、类、数值、字符串等**都被视为对象。
 - 函数可以像普通变量一样被赋值、传递、或作为返回值, 这体现了函数的“**一等公民 (first-class citizens)**”地位。

名称与对象的关系

- 变量名只是对内存中对象的**引用 (reference)**。
- 当执行赋值语句时 (例如 `x = 3`), Python 会将名称 `x` 绑定到一个对象 (此处为整数 3)。
- 可以通过 `type(x)` 查看对象类型。

函数的赋值与重绑定

```
Python
1 x = 3
2 x([1,3,5]) # ❌ 报错: 'int' object is not callable
3 x = sum    # 将内建函数 sum 赋给 x
4 x([1,3,5]) # ✅ 输出 9
5 type(x)    # <class 'builtin_function_or_method'>
```

- 说明: 变量名 `x` 可以从指向整数变为指向函数对象。
- 因此函数可以像数据一样被动态操作、重绑定。

按名称动态调用函数

- 可以使用 `getattr()` 根据函数名字符串从模块中获取函数对象:

```
Python
1 function = getattr(np, "mean")
2 function(np.array([1,2,3])) # 输出 np.float64(2.0)
```

- 这允许基于**字符串函数名**进行灵活的函数调用 (如反射机制)。

函数作为参数 (高阶函数)

- 函数可以被传入另一个函数作为参数或返回值:

```
Python
1 def apply_fun(fun, a):
2     return fun(a)
3
4 apply_fun(round, 3.5) # 输出 4
```

- 这种**接收或返回函数对象**的函数被称为 **higher-order functions (高阶函数)**。
- 许多内建函数 (如 `map`, `filter`, `reduce`) 正是此类高阶函数。

1.4 哪些操作是 function calls?

哪些操作是函数调用?

- Python 中一些语句 (statements) **不是函数调用**, 但会影响当前环境:

- `import`：导入模块或包
- `def`：定义函数或类
- `return`：从函数返回结果
- `del`：删除对象
- 虽然语法上不是函数调用，但它们本质上执行类似的动作。

• 运算符与面向对象函数调用（OOP）

- 运算符在底层对应类方法调用。例如：

```
Python
1 x = np.array([0,1,2])
2 x - 1          # array([-1, 0, 1])
3 x.__sub__(1)   # 与上式等价
4 x              # array([0, 1, 2]) （原对象未被修改）
```

- 运算符 `+`，`-`，`*`，`/` 等实质上是对对象方法（如 `__add__`，`__sub__`）的封装调用。
- 这体现了 **操作符重载（operator overloading）** 与 **泛型函数式（generic function OOP）** 的思想。

• 语法糖与解释器行为

- 表面上可写为 `return(x)` 或 `del(x)`，但解释器实际上将其解析为：
 - `return x`
 - `del x`
- 圆括号仅是语法允许的“**装饰性符号**”，并不改变解释器的底层解析逻辑。

| 1.5 Map Operations（映射操作）

• 定义与作用

- **映射（map）操作**：接受一个函数，并将该函数依次作用于集合中每个元素。
- 类似数学中的“函数映射”概念，是函数式编程中最常用的操作之一。
- 优点：**代码简洁、可读性高、避免显式循环**。

• Python 中的映射类型函数

- Python 提供多种 **map 类函数**：
 - 内置函数 `map()`
 - `pandas.DataFrame.apply()`
- 它们都是 **高阶函数（higher-order functions）**，即以函数作为参数。
- **列表推导式（list comprehension）** 也是映射操作的一种形式：

```
Python
1 x = [1, 2, 3]
2 y = [pow(val, 2) for val in x]
3 # [1, 4, 9]
```

• 基本用法：`map()`

- `map()` 会在可迭代对象（iterable）的所有元素上运行指定函数。
- 可迭代对象包括 `list`，`range()` 以及其他返回可迭代结构的函数。

```

Python
1 x = [1.0, -2.7, 3.5, -5.1]
2 list(map(abs, x))
3 # [1.0, 2.7, 3.5, 5.1]

```

- `map()` 还支持 多个可迭代对象：

```

Python
1 list(map(pow, x, [2, 2, 2, 2]))
2 # [1.0, 7.29, 12.25, 26.01]

```

• 使用匿名函数 (lambda)

- 可通过 `lambda` 函数 即时定义一个临时函数，无需命名：

```

Python
1 x = [1.0, -2.7, 3.5, -5.1]
2 result = list(map(lambda val: val * 2, x))
3 # [2.0, -5.4, 7.0, -10.2]

```

- `lambda` 函数 是一种 匿名函数 (anonymous function)，用于一次性的小操作。

• 使用 `functools.partial` 预设参数

- 如果要为函数预设部分参数，可用 `partial()` 创建“半固定函数”：

```

Python
1 from functools import partial
2 round3 = partial(round, ndigits=3)
3 list(map(round3, [32.134234, 7.1, 343.7775]))
4 # [32.134, 7.1, 343.777]

```

• 与 for 循环的对比

- `Pandas` 的 `apply` 方法 支持将函数直接映射到分组数据上：

```

Python
1 # Stratified analysis 示例
2 subsets = df.groupby('grouping_variable')
3
4 # map风格 (apply): 简洁且易读
5 results = subsets.apply(analysis_function)
6
7 # for 循环写法: 更冗长
8 results = []
9 for _, subset in subsets:
10     results.append(analysis_function(subset))

```

- 相比之下，`apply()` 风格更加函数式，简洁且更符合数据分析流程。

• 延伸：MapReduce 框架

- 映射操作是 **MapReduce** 的核心机制：
 - “Map” 阶段：并行地将函数作用于数据块。
 - “Reduce” 阶段：聚合映射结果。
- 被广泛应用于 **Hadoop** 和 **Spark** 等大数据平台中，用于分布式数据处理。

| 2 Function evaluation, frames, and the call stack (函数求值, 帧和调用栈)

| 2.1 概述

- **概览 (Overview)**
 - 在程序执行过程中，函数往往会在其他函数内部被调用，从而形成一个 **嵌套调用链**。
 - 这些函数调用按顺序被压入和弹出，组成了 **调用栈 (call stack)**。
 - 可以将调用栈想象为“**食堂托盘堆**”：
 - 当一个函数被调用时，它被 **压入栈顶 (push)**；
 - 当该函数执行完毕时，它被 **弹出栈 (pop)**。
-
- **调用栈在调试中的作用**
 - 理解调用栈对于阅读错误信息和调试至关重要。
 - 在 Python 中，当错误发生时，会显示完整的 **调用栈追踪 (traceback)**，能帮助我们定位错误来源。
 - 优点：提供了完整的函数调用历史，有助于理解错误的上下文。
 - 缺点：输出可能非常冗长，难以快速阅读。
 - 对比 R：
 - R 默认只显示错误发生的函数。
 - 若需查看完整调用链，可使用 `traceback()`。
-

- **Python 函数求值 (Function Evaluation) 流程**
 1. **参数求值与匹配**
 - 用户传入的实参会先在 **调用作用域 (calling scope)** 中求值。
 - 计算结果与函数定义中的形参名进行 **匹配绑定**。
 2. **创建新帧 (Frame) 与命名空间 (Namespace)**
 - 调用函数时，Python 会为该调用创建一个新的 **执行帧 (frame)**，并分配一个独立的 **局部命名空间 (local namespace)**。
 - 该帧会被 **压入调用栈 (push onto the stack)**。
 - 函数的参数（包括默认参数）会在此命名空间中赋值。
 3. **函数体求值 (Evaluation in local scope)**
 - 函数体的执行发生在 **局部作用域** 中。
 - 若在局部找不到某个变量名，Python 会根据 **词法作用域 (lexical scoping)** 规则依次查找：
 1. 当前局部作用域 (local scope)
 2. 外层嵌套函数作用域 (enclosing scopes, if any)
 3. 全局作用域 (global/module scope)
 4. 内建作用域 (built-in scope)
 4. **函数返回与栈清理**
 - 函数执行结束后，`return` 的值会被传回调用点所在的作用域。
 - 当前帧被 **弹出调用栈 (popped off the stack)**。
 - 若该命名空间不是其他活动作用域的外层，则会被销毁（释放内存）。

| 2.2 Frames and the call stack (帧和调用栈)

- **基本概念**

- Python 会持续追踪当前的 **调用栈 (call stack)**。
- 每次函数调用时，系统会创建一个 **帧 (frame)**，其中包含该函数调用的 **局部命名空间 (local namespace)**，即当前函数中的局部变量。
- 这些帧按调用顺序被压入栈中 (push)，当函数结束时再被弹出 (pop)。

• 访问与查看调用栈

- Python 提供多种方法来**查询栈中的帧信息**并访问其中的对象。
- 借助 `traceback` 模块，可以直接查看当前调用栈的完整结构。
- 常用函数：
 - `traceback.print_stack()`：打印当前调用栈。
 - `traceback.format_stack()`：返回调用栈信息的字符串列表，适合程序化处理。

• 示例：使用 `traceback` 打印调用栈

```
Python
1  import traceback
2
3  def function_a():
4      function_b() # line 2 within function, line 4 overall
5
6  def function_b():
7      # some comment
8      # another comment
9      function_c() # line 4 within function, line 9 overall
10
11 def function_c():
12     traceback.print_stack() # line 2 within, line 12 overall
13     # raise RuntimeError("A fake error")
14
15 function_a() # line 1 relative to the call, line 15 overall
```

执行结果（交互模式）

```
1  File "<stdin>", line 1, in <module>
2  File "<stdin>", line 2, in function_a
3  File "<stdin>", line 4, in function_b
4  File "<stdin>", line 2, in function_c
```

- 说明：
 - 栈顶是当前执行的函数 `function_c`。
 - 每一层函数的调用顺序都被完整记录。

• 在脚本中运行（例如 `trace.py`）

```
1  File "file.py", line 15, in <module>
2      function_a()
3  File "file.py", line 4, in function_a
4      function_b()
5  File "file.py", line 9, in function_b
6      function_c()
7  File "file.py", line 12, in function_c
8      traceback.print_stack()
```

- 此时行号对应的是脚本中的绝对位置（相对于整个文件）。
- 可以清晰看到调用链的层次结构。

- 错误追踪 (Traceback 与异常)

- 若将 `traceback.print_stack()` 注释掉, 并改为:

```
Python
1 raise RuntimeError("A fake error")
```

- 输出结果会显示相同的调用栈信息。
- 这正是 Python 在抛出异常时显示的 **traceback**, 即从错误发生处向上追溯至最初调用点的完整路径。

3 函数输入和输出

3.1 Arguments (函数输入)

- 查看函数参数与默认值

- 可使用 **help** 系统 或 `?function_name` (适用于 R 语言) 查看函数的参数信息与默认值。
- 示例:

```
Python
1 def add(x, y, z=1, absol=False):
2     if absol:
3         return abs(x + y + z)
4     else:
5         return x + y + z
```

- 参数定义规则

- 定义函数时, 无默认值参数必须位于默认值参数之前。
- 示例调用:

```
Python
1 add(3, 5) # 9
2 add(3, 5, 7) # 15
3 add(3, 5, absol=True, z=-5) # 3
4 add(z=-5, x=3, y=5) # 3
```

- 若缺少必要位置参数会报错:

```
Python
1 add(3)
2 # TypeError: add() missing 1 required positional argument: 'y'
```

- 位置参数与关键字参数

- 位置参数 (**positional arguments**): 按顺序提供。
- 关键字参数 (**keyword arguments**): 使用 `name=value` 形式指定。
- 规则: 位置参数必须在关键字参数之前。

```
Python
1 add(z=-5, 3, 5)
2 # SyntaxError: positional argument follows keyword argument
```


- 可变数量的参数: `*args` 与 `**kwargs`
 - 使用 `*args` 表示不定数量的位置参数。
 - `args` 通常是一个 `tuple`, 可以被遍历或求和。

```
Python
1 def sum_args(*args):
2     print(args[2])      # 第三个参数
3     total = sum(args)
4     return total
5
6 result = sum_args(1, 2, 3, 4, 5)
7 # 输出:
8 # 3
9 # 15
```

- 应用示例: `os.path.join` 可接收任意数量参数或解包列表:

```
Python
1 os.path.join('a', 'b', 'c') # 'a/b/c'
2 x = ['a', 'b', 'c']
3 os.path.join(*x)           # 'a/b/c'
```

- 若要同时接收关键字参数, 需定义:

```
Python
1 def func(*args, **kwargs):
2     ...
```

- `args` 是 `tuple`, `kwargs` 是 `dict`。

3.2 Function Outputs (函数输出)

- 基本返回语句
 - `return x` 用于指定函数输出, 可在函数中任意位置出现。
 - 当 `return` 执行时, 函数立即退出。

- 多个返回值
 - 可返回多个对象, 返回结果会被打包成 `tuple`。

```
Python
1 def f(x):
2     if x < 0:
3         return -x**2
4     else:
5         res = x^2 # 注意: 这是按位异或, 不是幂运算
6         return x, res
7
8 f(-3) # -9
9 f(3)  # (3, 1)
10
11 out1, out2 = f(3)
```

- 可通过解包（unpacking）将多个输出赋值给不同变量。

- **无返回值函数**

- 若函数主要用于副作用（如打印、修改外部状态等）：
 - 可省略 `return` ；
 - 或显式写作 `return None` ；
 - 或仅写 `return` （效果相同）。

| 4 Pass by Value vs. Pass by Reference

| 4.1 Core Concepts

- **Pass-by-value（值传递）**

- 函数调用时，实参会被复制一份到函数内部。
- 函数中对参数的修改 **不会影响** 原始对象。
- 特点：
 - 模块化、无副作用（side effects）；
 - 但可能占用更多内存、计算效率低。

- **Pass-by-reference（引用传递）**

- 传入函数的不是副本，而是 **指向原对象的引用**。
- 函数内部修改对象内容时，会直接影响函数外部的状态。
- 特点：
 - 高效（避免复制大型对象）；
 - 但更“危险”，可能引入隐藏的副作用。

🔗 Logic ▾

Pass-by-value 更符合函数式编程（functional programming）的思想：
函数的效果仅体现在返回值，而非对外部状态的修改。

| 4.2 Python 的行为

- Python 中，**数组与其他可变对象（mutable types）** 采用 **引用传递（pass-by-reference）**。
- 不可变对象（immutable types），如 **tuple**、**int**、**str**，即便传引用，也不会被修改。



Python

```
1 def myfun(x):
2     x[1] = 99
3
4 y = [0, 1, 2]
5 z = myfun(y)
6 type(z)    # <class 'NoneType'>
7 y          # [0, 99, 2]
```

⚠ Remark ▾

尽管 Python 并非严格意义上的“pass-by-reference”，
但由于可变对象是共享引用的，因此表现上等价于引用传递。

4.3 示例分析：哪些修改会影响外部状态？

```
Python
1 def myfun(f_scalar, f_x, f_x_new, f_x_newid, f_x_copy):
2     print(id(f_scalar))
3     print(id(f_x))
4
5     f_scalar = 99          # 不影响外部 scalar
6     f_x[0] = 99           # 修改外部 x
7     f_x_new = [99, 2, 3]  # 不影响外部 x_new
8     y = f_x_newid
9     y[0] = 99             # 修改外部 x_newid
10    z = f_x_copy.copy()
11    z[0] = 99             # 不影响外部 x_copy
```

```
Python
1 scalar = 1
2 x = [1,2,3]
3 x_new = [1,2,3]
4 x_newid = [1,2,3]
5 x_copy = [1,2,3]
6
7 myfun(scalar, x, x_new, x_newid, x_copy)
```

- 状态保留（未修改）

```
1 scalar = 1
2 x_new = [1, 2, 3]
3 x_copy = [1, 2, 3]
```

- 状态被修改

```
1 x = [99, 2, 3]
2 x_newid = [99, 2, 3]
```

Logic ▾

若在函数中**重新赋值（rebind）**一个变量名，则创建的是新的局部引用，不会修改外部状态。
若仅**修改对象内容（mutate）**，则会影响外部对象。

4.4 Mutable 对象的通性

- 与 `list` 类似，**NumPy 数组**等可变对象在函数中被修改时，外部状态同样会受到影响。

4.5 Pointers（指针, optional）

Logic ▾

理解“引用传递”的底层机制，可从 C 语言中的 **指针（pointer）** 理解。

在 C 中：

```

C
1 int x = 3;
2 int* ptr;
3 ptr = &x;    // 获取 x 的地址
4 *ptr * 7;    // 解引用, 返回 21

```

- `int* ptr`: 声明 `ptr` 为指向整数的指针;
- `&x`: 获取 `x` 的内存地址;
- `*ptr`: 通过指针访问该地址的内容。

数组本质上也是指针:

```

C
1 int x[10]; // x 实际上是指向数组首元素的指针

```

可以通过 `x[0]` 或 `*x` 访问首元素。

函数中使用指针的效果

```

C
1 int myCal(int* ptr){
2     *ptr = *ptr + *ptr;    // 原地修改对象
3 }
4
5 myCal(&x);    // 传入 x 的地址, x 会被修改

```

⚠ Remark ▾

在 C 中, 函数接收的是 **地址 (指针)**, 因此函数内部可直接修改外部对象的值。

Python 的行为与此类似:

它通过**对象引用 (object reference)** 传递参数, 从而在函数中可以“间接地”修改原对象 (尤其是 mutable 类型)。

| 5 Namespaces 和 Scopes

| 5.1 Namespaces

Namespace (命名空间) 是从 names 到 objects 的 mapping, 允许 Python 通过 name 找到 object

Namespace 在执行 Python 代码的过程中创建和移除:

- 当函数运行时, 会为函数中的 local variables 创建一个 namespace
- 在函数执行完成时删除 namespace
- 每个的函数调用有单独的 namespace

| 5.2 Scope

Scope (作用域) 决定了代码中的某个位置可以访问哪些 namespaces

- Scope 是嵌套的
- 决定 Python 以何种顺序搜索各种 namespaces 以查找对象

| 5.3 Key Scopes

关键作用域, 按搜索 namespaces 的顺序排序 (LEGB):

- **Local scope 局部作用域**: given function/class method 内可用的 objects
- **Non-local (Enclosing) scope 非局部作用域**: 包含 given function 的函数中的可用的 objects
- **Global (Module) scope 全局作用域**: 定义 given function 的 module 中可用的 objects

⚠ Remark ▾

若代码不是在函数内部执行, global scope 将可以被当作 local scope

- **Built-ins scope 内置作用域**: 通过 Python 提供的 built-ins module 中可用的 objects, 可从任何地方访问

⚠ Remark ▾

- `import` 将导入 module 的 name 添加到 current (local) scope 的 namespace
- 可以使用 `locals()` 和 `globals()` 查看局部和全局命名空间.



Shell

```
1 cat local.py
```

```
1 gx = 7
2
3 def myfun(z):
4     y = z*3
5     print("local: ", locals())
6     print("global: ", globals())
```



Python

```
1 import local
2
3 gx = 99
4 local.myfun(3)
```

```
1 local: {'z': 3, 'y': 9}
2 global: {'__name__': 'local', '__doc__': None, '__package__': '', '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x1038e3d30>, ..., 'gx': 7, 'myfun':
<function myfun at 0x103903010>}
```

5.4 Lexical scoping 和 enclosing scopes

Enclosing scope 是定义函数的 scope, 而不是调用函数的 scope

一旦搜索了 enclosing scope, 如果找不到对象名称, 那么 Python 会在**定义函数**的 global/module scope 中查找, 而不是从调用它的地方查找

这种方法称为**lexical scoping**. R 和许多其他语言也使用 lexical scoping

5.4.1 词法作用域示例

案例 1:



Python

```
1 x = 3
2 def f2():
3     print(x)
4 def f():
5     x = 7
```

```
6     f2()
7     f() # 输出 3
```

案例 2:

```
Python
1 x = 3
2 def f2():
3     print(x)
4 def f():
5     x = 7
6     f2()
7 x = 100
8 f() # 输出 100
```

案例 3:

```
Python
1 x = 3
2 def f():
3     def f2():
4         print(x)
5         x = 7
6         f2()
7 x = 100
8 f() # 输出 7
```

案例 4:

```
Python
1 x = 3
2 def f():
3     def f2():
4         print(x)
5     f2()
6 x = 100
7 f() # 输出 100
```

案例 5:

```
Python
1 y = 100
2 def fun_constructor():
3     y = 10
4     def g(x):
5         return x + y
6     return g
7
8 ## fun_constructor() creates functions
9 myfun = fun_constructor()
10 myfun(3) # 输出 13
```

1. What is the enclosing scope for the function `g()` ?

The local scope of `fun_constructor()`

2. Which `y` does `g()` use?

10

3. Where is `myfun` defined (this is tricky – how does `myfun` relate to `g`)?
`myfun` is simply a reference to `g`
4. What is the enclosing scope for `myfun()` ?
The enclosing scope of `myfun()` is still the local scope of `fun_constructor()` where `g` was created.
5. When `fun_constructor()` finishes, does its scope (and namespace) disappear? What would happen if it did?
Normally the scope would disappear, but because `g` is a closure, Python preserves the needed variables (`y = 10`). If the scope vanished completely, calling `myfun()` would raise a `NameError` .
6. What does `myfun` use for `y` ?
`myfun` uses the closed-over `y = 10` .

5.5 Global 和 non-local variables

- 可以使用 `global` 创建和修改 global variable 中的变量
- 可以使用 `nonlocal` 创建和修改 enclosing scope 中的变量

```
Python
1  del x
2  def myfun():
3      global x
4      x = 7
5
6  myfun()
7  print(x)  # 7
8
9  x = 9
10 myfun()
11 print(x)  # 7
12
13 def outer_function():
14     x = 10
15     def inner_function():
16         nonlocal x
17         x = 20
18         print(x)  # 10
19     inner_function()
20     print(x)  # 20
21
22 outer_function()
```

⚠ Remark ▾

在 R 中, 可以使用全局赋值运算符 `<-` 做类似的事情

5.6 Closures 闭包

🔗 Logic ▾

将 data 与 function 绑定的一种方法是使用 `closure`. 这是一种 functional programming 的方式, 可以实现类似于 OOP 中的 class 的东西

Closure 的实现方法:

1. 在函数调用中创建一个或多个函数, 并将函数作为输出返回
2. 当执行原始函数时, 新函数被创建并返回, 然后可以调用该函数
3. 该函数可以访问 enclosing scope 中的 objects

使用 Closure 的优点:

区别于使用 global variable, closure 中的 data 与 functions 被绑定在一起, 并受到保护, 不会被用户更改

```
Python
1 x = np.random.normal(size=5)
2 def scaler_constructor(data):
3     def g(param):
4         return param * data
5     return g
6
7 scaler = scaler_constructor(x)
8 del x # 演示我们不再需要 x
9
10 scaler(3)
11 # array([-4.5020648 , -3.25168752, -4.05046623,  0.66985868,  3.50796174])
12
13 scaler(6)
14 # array([-9.0041296 , -6.50337504, -8.10093246,  1.33971736,  7.01592347])
```

⚠ Remark ▾

调用 `scaler(3)` 会将 3 乘以存储在函数 `scaler` 的 closure (the namespace of the enclosing scope) 中的 data 值

这是一个更实际的例子:

```
Python
1 def make_container(n):
2     x = np.zeros(n)
3     i = 0
4     def store(value=None):
5         nonlocal x, i
6         if value is None:
7             return x
8         else:
9             x[i] = value
10            i += 1
11    return store
12
13 nboot = 20
14 bootmeans = make_container(nboot)
15
16 import pandas as pd
17 iris = pd.read_csv('https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/io/data/csv/iris.csv')
18 data = iris['SepalLength']
19
20 for i in range(nboot):
21     bootmeans(np.mean(np.random.choice(data, size=len(data), replace=True)))
22
23 bootmeans()
24
25 bootmeans.__closure__
```

Closures 也被用作 "function factories" (外侧的 generator function 被称为 "factory function"), 用来轻松生成一组相关函数, 下面是一个例子:


```

Python
1 def number_formatter(notation='US'):
2     """
3     Creates a closure for formatting decimal numbers.
4
5     Args:
6         notation (str): 'US' for US notation (commas for thousands, period for decimal)
7                         'EU' for European notation (periods for thousands, comma for decimal)
8
9     Returns:
10        function: A closure that formats numbers according to the specified notation
11    """
12    def format_number(number):
13        ## GitHub Copilot suggested the `number:,` syntax and the string replace approach.
14        result = f"{number:,}"
15        if notation == 'US':
16            # US notation: 1,234.56
17            return result
18        elif notation == 'EU':
19            # European notation: 1.234,56
20            # Swap commas and periods
21            return result.replace(',', 'TEMP').replace('.', ',').replace('TEMP', '.')
22        else:
23            raise ValueError("Notation must be 'US' or 'EU'")
24
25    return format_number
26
27    us_printer = number_formatter('US')
28    eu_printer = number_formatter('EU')
29
30    us_printer(1234.56)      # '1,234.56'
31    eu_printer(1234.56)      # '1.234,56'

```

6 Decorators

- **基本概念**
 - **Decorator (装饰器)** 是一种 **函数包装器 (wrapper)**: 它在 **不修改原函数代码** 的前提下, **扩展函数的功能**。
 - 核心思想: 在函数调用的“前后”自动执行某些逻辑 (如打印、计时、检查权限、缓存等)。
 - 装饰器的底层是 **高阶函数 (higher-order function)** —— 接收函数作为参数并返回新函数。

- **手动创建一个简单装饰器**

```

Python
1 def verbosity_wrapper(myfun):
2     def wrapper(*args, **kwargs):
3         print(f"Starting {myfun.__name__}.")
4         output = myfun(*args, **kwargs)
5         print(f"Finishing {myfun.__name__}.")
6         return output
7     return wrapper
8
9    verbose_rnorm = verbosity_wrapper(np.random.normal)
10    x = verbose_rnorm(size=5)
11    x

```

输出:

```
1 Starting normal.
2 Finishing normal.
3 array([ 1.0741,  0.2012, -1.0965, -1.9303, -2.1164])
```

Logic ▾

`verbosity_wrapper` 接收函数 `myfun` 并返回一个新的函数 `wrapper`。
在调用时：

1. 执行额外逻辑（打印信息）；
2. 调用原始函数；
3. 返回原函数结果。

这种“函数返回函数”的结构依赖于 **闭包 (closure)**，使得 `wrapper` 能访问 `myfun`。

• 使用语法糖 @ 语法简化装饰器



Python

```
1 @verbosity_wrapper
2 def myfun(x):
3     return x
4
5 y = myfun(7)
```

输出：

```
1 Starting myfun.
2 Finishing myfun.
3 7
```

等价于：



Python

```
1 myfun = verbosity_wrapper(myfun)
```

Logic ▾

装饰器语法糖（`@decorator_name`）在函数定义时立即生效。
这意味着函数名 `myfun` 实际上被替换为包装后的版本。

• 常见应用场景

- **调试与日志**：记录函数的输入、输出、执行时间。
- **性能监控**：统计函数运行耗时。
- **访问控制**：验证权限或状态。
- **缓存 (memoization)**：保存函数结果以减少重复计算。
- **异步/并行执行**：如 `@dask.delayed` 或 `@numba.jit`。

• 示例：计时装饰器



Python

```
1 import time
```

```

2
3 def timer(func):
4     def wrapper(*args, **kwargs):
5         start = time.time()
6         result = func(*args, **kwargs)
7         end = time.time()
8         print(f"{func.__name__} took {end - start:.4f} seconds.")
9         return result
10    return wrapper
11
12 @timer
13 def slow_function():
14     time.sleep(1)
15     return "Done"
16
17 slow_function()
18 # 输出: slow_function took 1.000x seconds.

```

⚠ Remark ▾

使用装饰器后，函数的元数据（如 `__name__`、`__doc__`）会被包装函数覆盖。
为保持原信息，应使用 `functools.wraps()`：



Python

```

1 from functools import wraps
2
3 def timer(func):
4     @wraps(func)
5     def wrapper(*args, **kwargs):
6         ...
7     return wrapper

```

否则 `func.__name__` 会显示为 `"wrapper"` 而非原函数名。

• 装饰器的高级特性

- **带参数的装饰器**：装饰器本身再被一个函数包装，用于接收配置参数。



Python

```

1 def repeat(n):
2     def decorator(func):
3         def wrapper(*args, **kwargs):
4             for _ in range(n):
5                 func(*args, **kwargs)
6             return wrapper
7         return decorator
8
9 @repeat(3)
10 def greet():
11     print("Hello!")

```

输出：

```

1 Hello!
2 Hello!
3 Hello!

```

- **装饰器链 (decorator stacking)**:
多个装饰器可叠加执行，按自下而上的顺序调用。

- **现实中的装饰器示例**

- **Dask 并行计算**:
`@dask.delayed` 将普通函数转为延迟计算任务。
- **Numba JIT 编译**:
`@numba.jit` 将 Python 函数即时编译为高性能机器码。
- **Flask 路由注册**:
`@app.route('/home')` 绑定 URL 与处理函数。

Logic ▾

装饰器的强大之处在于——
它让函数的“行为增强”可以独立定义、复用、组合，
从而实现更简洁、更模块化的程序结构。