第 9 讲 · 讲义

类（一）：对象与封装

课件的每一页，配上讲这一页时说的话。幻灯片是网页原生渲染的，文字可以选中、可以搜索，也可以直接打印。



第 1 页 · Objects and Encapsulation

> 今天我们一起实现一个小型双角色探险模拟器。先看图形演示，再用 Python 实现角色核心逻辑。图形界面按同样的状态规则显示行动，界面开发不属于本讲要求。最终每个角色都能保存自己的坐标和金币、移动、查询距离、付款和拾取物品。我们先用学过的字典与函数起步，再把同一份程序逐步组织成 Rover 类；每一个新机制都为下一个功能服务。

Our Project · Two-Rover Expedition

Build two rovers that can **move, pay and collect items**.



Start: a (0, 0), b (3, 4)
Each: 100 coins, an empty bag

- 目标：a 能行动，b 保留自己的状态。

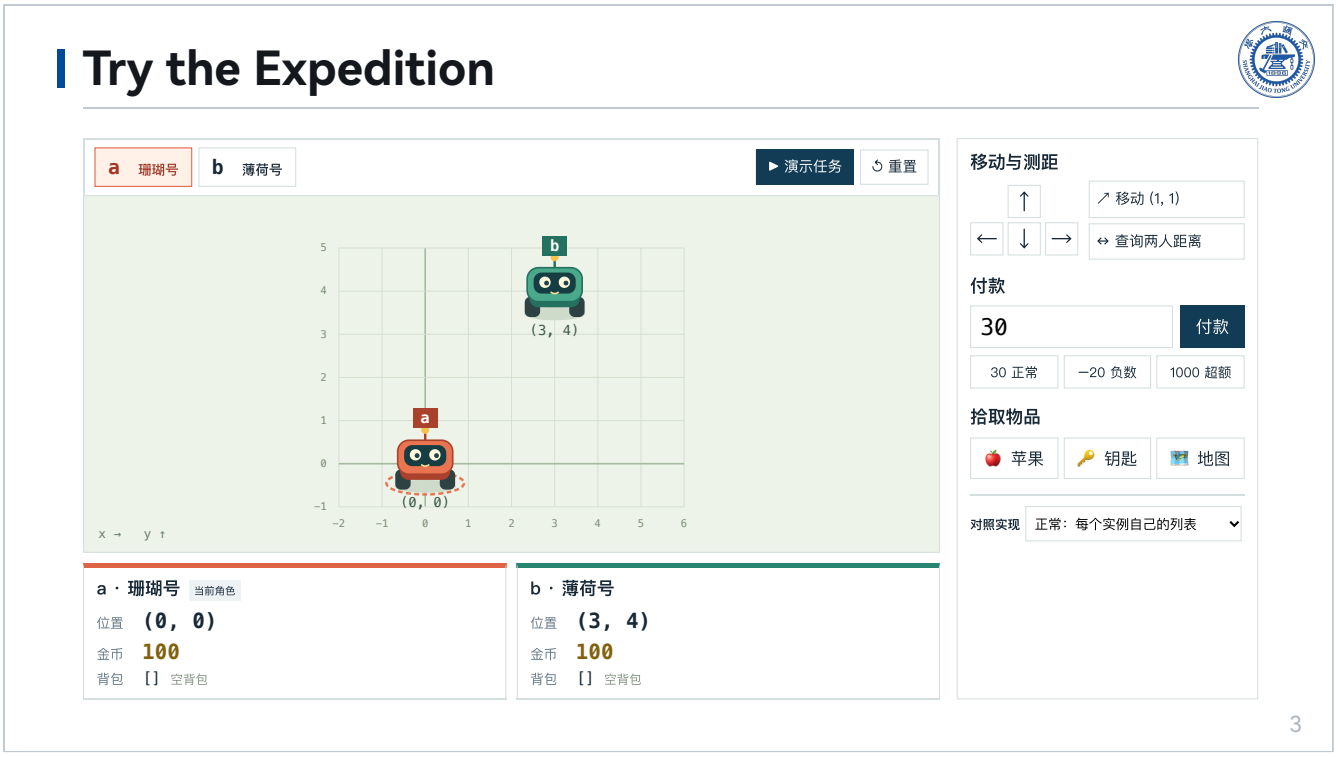
One mission 一次探险

a moves by (1, 1)
a pays 30 coins
a picks an apple

Must stay independent

b keeps its position,
100 coins and an empty bag

> 先明确成品是什么：两个角色 a、b 从零零和三四出发，各有一百金币和一个空背包。我们要让 a 移动一步、支付三十金币、捡起一个苹果。最后 a 应为一一、七十金币和一个苹果，b 应保持三四、一百金币和空背包。今天逐步实现的是这段可运行的角色模拟程序。下一页演示完整任务，随后从第一版开始实现。精确状态仍由角色逻辑维护。



第 3 页 · Try the Expedition

> 这是本讲最终要实现的双角色探险。选择 a，移动、支付三十金币并拾取苹果，观察 b 的状态保持。各阶段的游戏界面会随课程重新出现，分别验证位置、金币和独立背包。图形界面用于显示 Python 角色逻辑的结果。

Start with What We Know

A **dictionary** stores one rover; **functions** perform its actions.

Two records 两份角色记录

```
a = {"x": 0, "y": 0,
     "coins": 100, "items": []}
b = {"x": 3, "y": 4,
     "coins": 100, "items": []}
```

Target actions 待实现的函数接口

```
move(a, 1, 1)
spend(a, 30)
pick(a, "apple")
```

- 这些知识已经够写出来。接下来关注：功能增加后，怎样集中维护？

> 我们不需要先学类才能实现任务。字典能够把一个角色的数据放在一起，普通函数能够移动、扣钱和拾取。下载包中的零号版本就是完整正确的字典与函数实现。右边先展示我们计划的调用接口，下一页看继续扩展时会遇到的维护问题。

A New Feature, a Missed Field

After adding coins, one creation site still uses the **old record shape**.

Complete faulty example 完整错误示例

```
a = {"x": 0, "y": 0, "coins": 100}
b = {"x": 3, "y": 4}

def spend(rover, amount):
    rover["coins"] -= amount

spend(b, 30)
```

Python output

```
KeyError: 'coins'
```

- 创建角色的代码散落在多处；每加一个字段，就要核对所有入口。

> 坐标版原来只需要 x、y。现在添加金币，我们给 a 补上 coins，却漏改 b 的创建位置。取 b 的 coins 时实际抛 KeyError。程序扩展后，重复的创建代码容易不同步。用 make_rover 函数统一创建可以解决这一部分；我们还希望把角色的动作和状态规则也集中在同一接口里。

Procedural Programming

面向过程：围绕**任务步骤和函数**组织程序，
函数接收数据并完成相应操作。

Choose a function

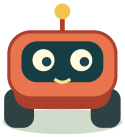
move

spend

pick

update a

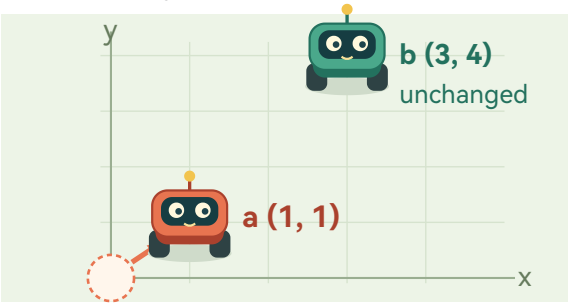
a · data



x = 0, y = 0

100 coins · []

Result · only a moves




```
move(a, 1, 1)
```

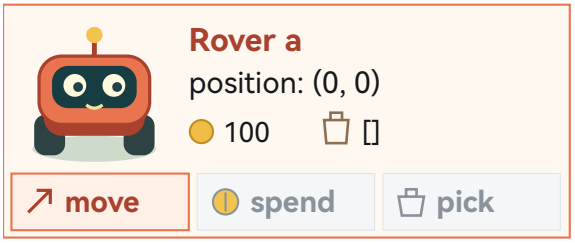
- 调用移动函数，传入需要处理的角色 a。

面向过程围绕任务步骤和函数组织程序。左侧先选中 move 函数，把角色 a 的数据作为参数传入，对应底部的 move(a,1,1)。右侧显示这一次调用的结果：a 从零零移动到一一，b 保持三四。移动、付款和拾取是不同的操作；本页只执行移动。角色数据和处理它的函数分别组织，函数通过参数确定当前处理哪份数据。

Object-Oriented Programming

面向对象：围绕**具有状态和行为的对象**组织程序，
通过对象的接口完成操作与协作。

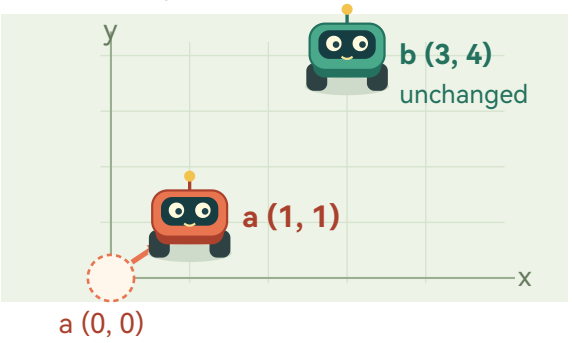
Choose the rover




```
a.move(1, 1)
```

- 选中角色 a，调用它提供的移动方法。

Result · only a moves



> 面向对象围绕具有状态和行为的对象组织程序。左侧的角色卡同时展示 a 保存的坐标、金币和背包，以及它提供的移动、付款、拾取接口。选中移动，对应底部的 a.move(1,1)。右侧与上一页保持相同输入和结果：a 移到一一，b 保持三四。角色卡展示调用者可用的状态与接口，方法实现由类共享，后面再分析它的存放和绑定。本页只执行移动。

Same Task, Two Designs

Both programs move **a** and preserve **b**.

Procedural · complete program

```
def move(r, dx, dy):
    r["x"] += dx
    r["y"] += dy
a = {"x": 0, "y": 0}
b = {"x": 3, "y": 4}
move(a, 1, 1)
print("a:", a["x"], a["y"])
print("b:", b["x"], b["y"])
```

Object-oriented · complete program

```
class Rover:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
a, b = Rover(0, 0), Rover(3, 4)
a.move(1, 1)
print("a:", a.x, a.y)
print("b:", b.x, b.y)
```

Actual output, both: `a: 1 1` | `b: 3 4`

> 左右是两个完整可运行程序。初始坐标、移动算法与结果相同：a 变成一一，b 保持三四。左边字典保存数据，类外函数接收记录；右边类定义初始化和移动，实例保存坐标，方法通过 self 访问它。请分别指出数据在哪里建立、操作在哪里定义、调用时怎样选中角色。后面加入金币和背包时，我们继续将相关状态与规则集中到 Rover 定义。

Languages and Programming Styles

A language can support **several programming styles**.

Procedural	Object-oriented	Multi-paradigm
C	Java	Python
Pascal	Smalltalk	C++

Python: functions, objects, or a mixture within one program.

Python documentation · programming styles

- 根据数据、操作和维护需求选择组织方式。

> C、Pascal 是面向过程的常见代表，Java、Smalltalk 是面向对象的常见代表。Python 和 C++ 支持多种范式，Python 可以写以函数为主或以对象为主的程序，也可以组合使用。固定的数据处理流程通常用函数组织很直接；多个需要持续保存状态的实体，适合考虑清晰的对象接口。具体选择取决于状态、操作和维护需求。本讲通过类学习对象与封装，继承与多态在下一讲继续。

Build It in Three Stages

Each stage adds a capability to the **same Rover class**.

1	**Position 坐标**	Create two rovers → move one → measure distance
2	**Coins 金币**	Add a field → update self → read state
3	**Backpack 背包**	Add picking → discover sharing → create independent bags

- 最后运行开场的探险任务，检查 a 和 b 的完整状态。

> 路线按功能需求排列。第一版让角色有坐标、会移动、能测距；第二版增加金币字段，观察方法怎样更新与读取它；第三版加背包，排查串包，最后串起来跑一次任务。每个版本都会用到初始化、self、持久属性和共享引用。本讲可下载的零、一、二、三号版本分别对应字典起点、位置版、金币版和完整探险版。

Stage 1 · Create a Rover

One **class** defines the interface; each **instance** is a character.



Class 类

Rover
Start with x and y

Instances 实例

a: (0, 0)
b: (3, 4)

> 第一步只实现位置，金币和物品先留在需求里，不一次塞进构造函数。Rover 类定义这一类角色的共同接口，a、b 是两个具体实例。就像同一种装配说明能建立多个角色，它们共享行为的定义，但保存各自的状态。接下来把坐标保存到实例中。

Anatomy of a Class

A class body contains **attribute assignments** and **method definitions**.

Complete definition

```
class Rover:
    unit = "grid"

    def __init__(self, x=0):
        self.x = x

    def move(self, dx):
        self.x += dx

a = Rover(3)
a.move(1)
```

Read the indentation

class Rover: starts the class.
unit belongs to the class.
def defines a method.
self.x belongs to an instance.
Unindented code uses the class.

- 类体里的 def 使用普通函数语法；调用时再绑定接收者。

> class 后面是类名，冒号后缩进的语句组成类体。unit 的赋值在类体中建立类属性；两个 def 定义方法。方法体再缩进一级。self.x 的赋值在方法执行时建立或更新当前实例的属性。回到最外层后，Rover(3) 创建实例，a.move(1) 调用实例方法。先区分类体赋值与方法体内的属性赋值，随后分别跟踪它们的执行。

When Does the Class Body Run?

The **class body runs once here**; initialization runs for each instance.

Complete runnable example

```
class Rover:
    print("class body")
    def __init__(self):
        print("initialize")
print("ready")
a = Rover()
b = Rover()
```

Python output

```
class body
ready
initialize
initialize
```

Mechanism

class statement → execute its body
def → create a function
Rover() → initialize a new rover

> 执行 class 语句时，类体从上到下运行，因此先打印 class body。def 在类命名空间中建立函数，函数体此时等待调用。类语句结束后打印 ready；两次 Rover() 各调用一次初始化，所以再打印两次 initialize。执行时机决定了列表和默认参数会创建几次。

The Class Is an Object



Rover names a class object; a names one of its instances.

Complete runnable example

```
class Rover:
    pass
Alias = Rover
a = Alias()
print(Alias is Rover)
print(type(a) is Rover)
print(type(Rover) is type)
print(isinstance(a, Rover))
```

Python output

```
True
True
True
True
```

Mechanism

Alias and Rover refer to one class.
a refers to an instance.
Classes also have a type.

> class 执行结束后，名字 Rover 绑定到新建立的类对象。类对象也可以赋给另一个名字 Alias，Alias() 与 Rover() 调用同一个类。type(a) 是 Rover，type(Rover) 是 type。isinstance(a, Rover) 检查 a 是否为 Rover 的实例；继承对这一判断的影响留到下一讲。

Store a Position

p.x attaches a coordinate to the object referenced by **p**.

Complete Python example

```
1  class Rover:
2      pass
3
4  x = 100
5  p = Rover()
6  p.x = 3
7  p.y = 4
8  print(p.x, p.y, x)
```

Python output

```
3 4 100
```

Read the result

先认清：x 与 p.x 保存在哪里。

第 15 页 · Store a Position

> 先用空 Rover 建立实例 p，再给它添加 x 和 y。外部变量 x 为一百，实例 p.x 为三，这两个名字属于不同绑定。普通 Rover 实例允许动态增加字段，但让每个调用者自己添加坐标仍会出现漏项，下一页就检查这个问题。

A Rover Without Coordinates

Manual setup lets one rover start **without a required field**.

Complete Python example

```python
class Rover:
    pass

p = Rover()
p.x = 3
q = Rover()
print(p.x)
print(q.x)
```

Python output

```
3
Traceback (most recent call
last):
  File "example.py", line 8,
in <module>
    print(q.x)
          ^^^
AttributeError: 'Rover'
object has no attribute 'x'
```


第 16 页 · A Rover Without Coordinates

和字典版的漏字段问题一样，p 被加上 x，q 却没有。第一次打印三，随后 q.x 实际抛 AttributeError。换成 class 之后问题不会自动消失；我们需要一个统一的初始化入口，每次创建时都建立坐标。

Initialize Rover · 1/5

Evaluate Rover(3, 4). **a is not bound** to the new instance yet.

Python · execution

```python
class Rover:
    def __init__(self, x, y):
        self.x = x
        self.y = y

a = Rover(3, 4)
print(a.x, a.y)
```

1/5 · next statement to execute

Method	locals	Caller
not running		a not bound

No instance yet

- 统一入口，为每个角色建立坐标。

第 17 页 · Initialize Rover · 1/5

开始创建一个位于三四的角色。先计算赋值右侧，a 还没有绑定到这个新对象。 箭头标示将执行的语句；return 帧表示该行已经完成并即将返回。Rover A 是逻辑对象标签。

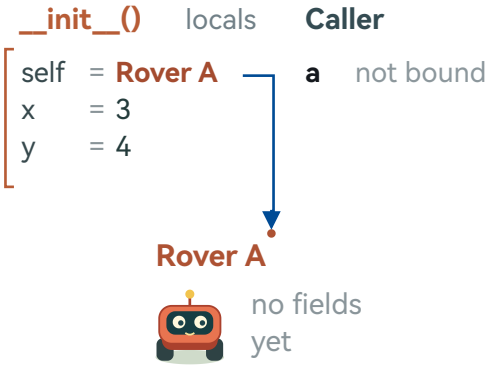
Initialize Rover · 2/5

The new instance exists. **self receives it**; x = 3 and y = 4.

Python · execution

```
1 class Rover:
2     def __init__(self, x, y):
▶ 3         self.x = x
4         self.y = y
5
6 a = Rover(3, 4)
7 print(a.x, a.y)
```

2/5 · next statement to execute



- 统一入口，为每个角色建立坐标。

18

第 18 页 · Initialize Rover · 2/5

新实例先被创建，再交给初始化方法，self 接收它，x、y 接收三和四。 箭头标示将执行的语句；return 帧表示该行已经完成并即将返回。Rover A 是逻辑对象标签。

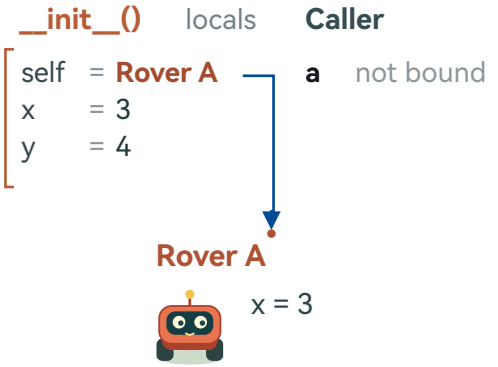
Initialize Rover · 3/5

After self.x = x, the rover has **x = 3**.

Python · execution

```
1 class Rover:
2     def __init__(self, x, y):
3         self.x = x
▶ 4         self.y = y
5
6 a = Rover(3, 4)
7 print(a.x, a.y)
```

3/5 · next statement to execute



- 统一入口，为每个角色建立坐标。

19

第 19 页 · Initialize Rover · 3/5

self.x = x 把本次调用中的局部坐标值保存到当前角色。 箭头标示将执行的语句；return 帧表示该行已经完成并即将返回。Rover A 是逻辑对象标签。

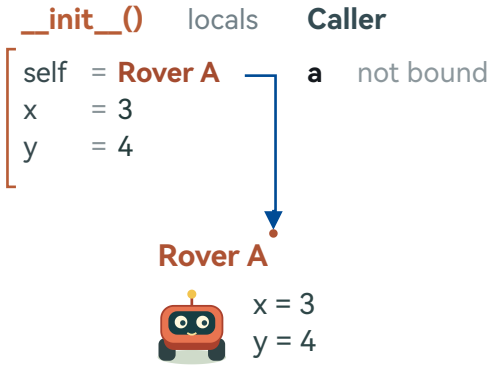
Initialize Rover · 4/5

After self.y = y, initialization ends. **__init__ returns None**.

Python · execution

```
1 class Rover:
2     def __init__(self, x, y):
3         self.x = x
▶ 4         self.y = y
5
6 a = Rover(3, 4)
7 print(a.x, a.y)
```

4/5 · return: highlighted line completed



- 统一入口，为每个角色建立坐标。

20

第 20 页 · Initialize Rover · 4/5

第二个字段也建立完成；初始化方法完成字段建立并返回 None；类调用随后返回实例。 箭头标示将执行的语句；return 帧表示该行已经完成并即将返回。Rover A 是逻辑对象标签。

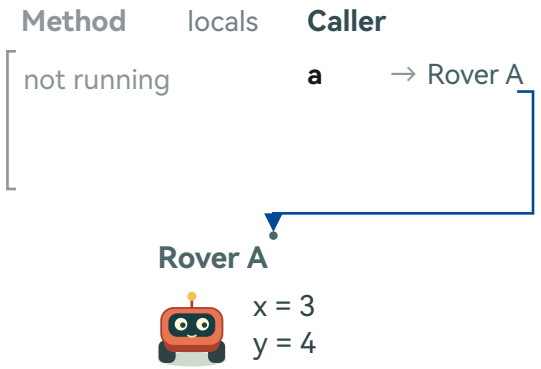
Initialize Rover · 5/5

The class call produces the rover. **Now a is bound to it**.

Python · execution

```
1 class Rover:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6 a = Rover(3, 4)
▶ 7 print(a.x, a.y)
```

5/5 · next statement to execute



- 统一入口，为每个角色建立坐标。

21

第 21 页 · Initialize Rover · 5/5

类调用的结果是实例，调用者的赋值才完成。现在每次 Rover(x,y) 都经过统一的字段建立过程。 箭头标示将执行的语句；return 帧表示该行已经完成并即将返回。Rover A 是逻辑对象标签。

Creation and Initialization

The **class call returns an instance**; __init__ must return None.

Complete runnable example

```
class Rover:
    def __init__(self):
        self.x = 0
        return self

a = Rover()
```

Exception

```
TypeError: __init__() should
return None, not 'Rover'
```

Mechanism

Rover() → create → initialize → result
__init__: set up fields; return None

> 类调用负责创建并返回实例，__init__ 接收已创建的实例、建立初始状态。一般省略 return，默认返回 None。这里错误地 return self，实际抛 TypeError，整个赋值未完成。可以用创建、初始化、绑定三个阶段理解前面的动画；__new__ 的定制留到后续课程。

Constructor Arguments

Initialization parameters follow **ordinary function argument rules**.

Complete runnable example

```
class Rover:
    def __init__(self, x=0, y=0):
        self.x, self.y = x, y
for a in [Rover(), Rover(3),
          Rover(y=4), Rover(3, 4)]:
    print(a.x, a.y)
```

Python output

```
0 0
3 0
0 4
3 4
```

Mechanism

Rover(3) → self, x=3, y=0
Rover(y=4) → self, x=0, y=4

> self 由调用机制传入，括号中的参数对应后面的 x、y。省略时使用默认零，Rover(3) 只提供 x，Rover(y=4) 使用关键字给 y，Rover(3,4) 同时提供两者。默认值在函数定义时求值；稍后用背包解释可变默认值的后果。

Where Do Attributes Live?

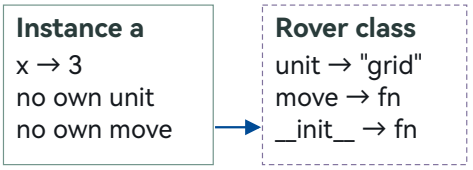
Instance state and the **class namespace** hold different bindings.

Complete runnable example

```
class Rover:
    unit = "grid"
    def __init__(self, x):
        self.x = x
    def move(self, dx):
        self.x += dx
a = Rover(3)
print(vars(a))
print(Rover.__dict__["unit"])
print("move" in Rover.__dict__)
print("move" in vars(a))
```

Python output

```
{'x': 3}
grid
True
False
```



a.unit: instance, then class

> vars(a) 显示普通实例自己的属性字典，此处只有 x。Rover.__dict__ 是查看类命名空间的只读映射，其中有 unit 与 move；方法函数保存在类命名空间，实例通过属性访问获得它。这个例子使用普通实例和普通字段，描述符、自定义属性访问和 slots 留待后续。

Two Rovers or Two Names?

Use a **new Rover(...) call** to create the second character.

Python excerpt

```
a = Rover(3, 4)
alias = a
b = Rover(3, 4)
alias.x = 8
print(a.x, b.x)
print(alias is a, b is a)
```

Python output

```
8 3
True False
```

What changed? 结果

alias is another name for a.
b is a separately created rover.

Full program 完整程序

- 坐标相同，不等于同一个角色。

> 统一创建之后，要建立两个独立角色，必须两次调用类。alias = a 只增加同一个角色的名字；通过 alias 改 x，a 也看到八，而另建的 b 仍为三。位置版同时使用默认参数，Rover() 从零零开始，Rover(3,4) 指定位置；默认值沿用普通函数规则。

Next Feature · Let a Move

`a.move(1, 1)` must change **a**, using the same shared method.

Add to Rover 添加到角色类

```
def move(self, dx, dy):
    self.x += dx
    self.y += dy
```

Receiver 接收者

a.move(1, 1)
↓ same receiver and arguments
Rover.move(a, 1, 1)

- 新问题：一份 move 代码，怎么选中当前角色？

> 角色现在有坐标，下一项需求是移动一个角色。我们把 move 放进同一个 Rover 定义。通过 a 访问普通方法会把 a 自动传给第一个参数，惯用名 self；dx、dy 是位移。类中只有一份方法实现，调用要靠接收者确定当前修改哪份坐标。

Defining and Calling Methods

A method can **update state**, **return a value**, or do both.

Complete runnable example

```
class Rover:
    def __init__(self, x=0):
        self.x = x
    def move(self, dx):
        self.x += dx
    def position(self):
        return self.x
a = Rover()
result = a.move(3)
print(result, a.position())
```

Python output

```
None 3
```

Mechanism

a.move(3) → Rover.move(a, 3)
a.position() → Rover.position(a)

27

第 27 页 · Defining and Calling Methods

> 这里给出包含定义和调用的完整最小类。类体内用 def 定义实例方法，第一个形参 self 接收当前实例。a.move(3) 将三传给 dx，更新 a.x，省略 return 所以得到 None。a.position() 返回当前坐标。读取返回值和检查字段变化是两项独立检查。

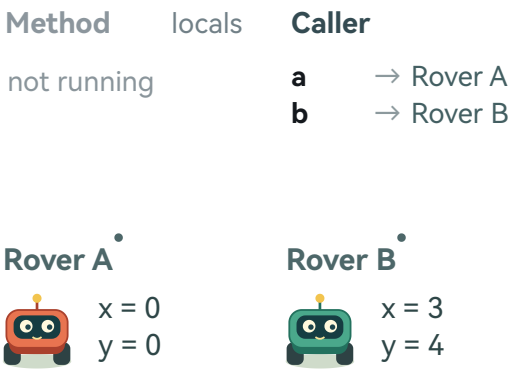
Run a.move · 1/5

Before the call: **a is at (0, 0)**; b is at (3, 4).

Python · execution

```
 5 def move(self, dx, dy):
 6     self.x += dx
 7     self.y += dy
 8
 9 a = Rover(0, 0)
10 b = Rover(3, 4)
▶ 11 result = a.move(1, 1)
12 print(a.x, a.y, b.x, b.y, result)
```

1/5 · next statement to execute



- self 选中角色，再更新它的字段。

28

第 28 页 · Run a.move · 1/5

> 建立两个测试角色，从零零与三四出发，只对 a 发出移动命令。 代码和对象字段来自本次真实执行。

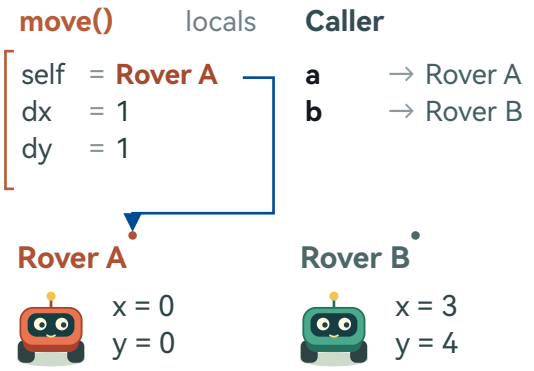
Run a.move · 2/5

Enter move: **self refers to a**, dx = 1, dy = 1.

Python · execution

```
  5 def move(self, dx, dy):
▶ 6     self.x += dx
  7     self.y += dy
  8
  9 a = Rover(0, 0)
 10 b = Rover(3, 4)
 11 result = a.move(1, 1)
 12 print(a.x, a.y, b.x, b.y, result)
```

2/5 · next statement to execute



- self 选中角色，再更新它的字段。

29

第 29 页 · Run a.move · 2/5

> 进入同一份 move 实现，self 绑定到 a，因此接下来的字段读取和写入都通过 a。 代码和对象字段来自本次真实执行。

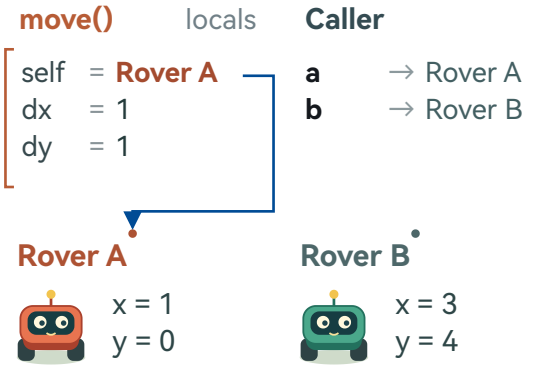
Run a.move · 3/5

The first update changes only **a.x from 0 to 1**.

Python · execution

```
  5 def move(self, dx, dy):
  6     self.x += dx
▶ 7     self.y += dy
  8
  9 a = Rover(0, 0)
 10 b = Rover(3, 4)
 11 result = a.move(1, 1)
 12 print(a.x, a.y, b.x, b.y, result)
```

3/5 · next statement to execute



- self 选中角色，再更新它的字段。

30

第 30 页 · Run a.move · 3/5

> 第一行完成后 a 的横坐标变为一，纵坐标仍为零，b 没有变化。 代码和对象字段来自本次真实执行。

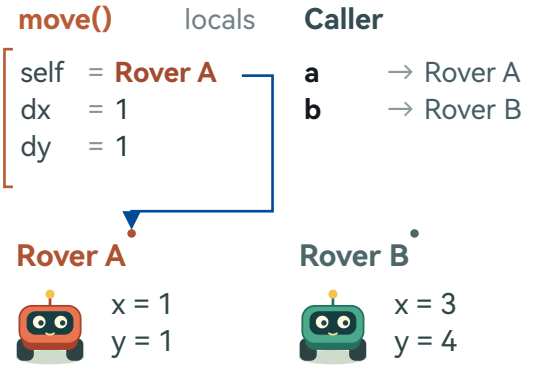
Run a.move · 4/5

The second update leaves a at (1, 1). **move returns None**.

Python · execution

```
 5 def move(self, dx, dy):
 6     self.x += dx
▶ 7     self.y += dy
 8
 9 a = Rover(0, 0)
10 b = Rover(3, 4)
11 result = a.move(1, 1)
12 print(a.x, a.y, b.x, b.y, result)
```

4/5 · return: highlighted line completed



- self 选中角色，再更新它的字段。

第 31 页 · Run a.move · 4/5

> 第二行完成后 a 为一一，b 仍为三四；方法没有显式 return，返回 None。 代码和对象字段来自本次真实执行。

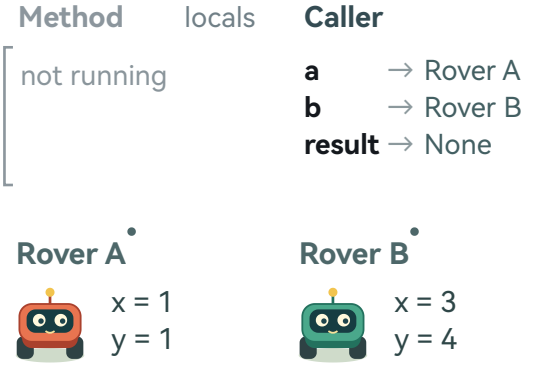
Run a.move · 5/5

Back in the caller: result is None; **a keeps its new position**.

Python · execution

```
 5 def move(self, dx, dy):
 6     self.x += dx
 7     self.y += dy
 8
 9 a = Rover(0, 0)
10 b = Rover(3, 4)
11 result = a.move(1, 1)
▶ 12 print(a.x, a.y, b.x, b.y, result)
```

5/5 · next statement to execute



- self 选中角色，再更新它的字段。

第 32 页 · Run a.move · 5/5

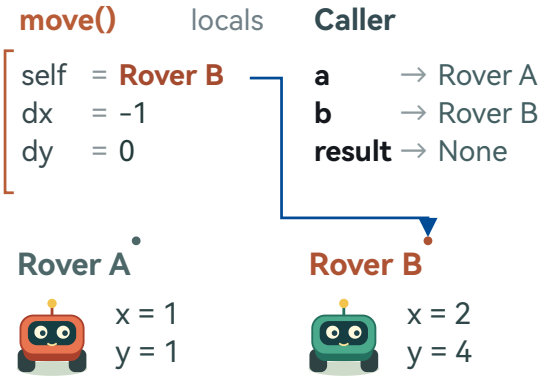
> 回到发出命令的位置，result 是 None，角色的新坐标则保留在实例中。这正是原地移动的接口。 代码和对象字段来自本次真实执行。

Can b Use the Same Method?

Call `b.move(-1, 0)`: now **self refers to b**.

Continue the receiver test 接续调用测试

```
b.move(-1, 0)
# self → b
# dx = -1, dy = 0
print(a.x, a.y, b.x, b.y)
```



- 测试结果：a = (1, 1)，b = (2, 4)。

33

第 33 页 · Can b Use the Same Method?

> 继续这个调用测试，让 b 向左移动。还是同一份 move，self 改为 b，所以 b 变为二四，a 保持一一。两个实例没有分别复制一份方法。这里是在验证方法选中正确接收者，最后的探险任务会从新建的两位角色开始。

A Bound Method Has Two Parts

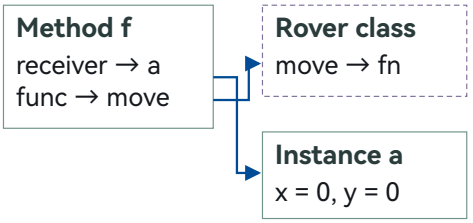
Accessing **a.move** combines a function with the receiver a.

Excerpt · position version

```
a = Rover()
b = Rover(3, 4)
f = a.move
print(f.__self__ is a)
print(f.__func__ is Rover.move)
print(a.move.__func__ is
      b.move.__func__)
```

Python output

```
True
True
True
```



Complete runnable example

> 类中的普通 Python 函数经实例访问后成为绑定方法。f.__self__ 是 a，f.__func__ 是类中保存的 Rover.move。a.move 与 b.move 的接收者不同，但 __func__ 相同。调用 f(1,1) 时，绑定的实例会成为该函数的第一个实参。

Store a Method, Call It Later

A stored **bound method keeps its receiver**.

Excerpt · position version

```
a = Rover()
action = a.move
action(1, 1)
print(a.x, a.y)
print(action.__self__ is a)
```

Python output

```
1 1
True
```

Mechanism

action(1, 1) calls Rover.move(a, 1, 1).
The receiver stays attached.

Complete runnable example

35

第 35 页 · Store a Method, Call It Later

> 方法也是可保存和传递的值。action = a.move 保存带有 a 的绑定方法，之后调用 action(1,1) 仍修改 a 的坐标。回调常用这种写法，例如界面按钮保存一个对象的方法。本例只演示已经学过的属性访问和调用，无需引入界面框架。

Count the Arguments

a.move(1, 1) supplies three arguments to the underlying function.

Complete runnable example

```
class Rover:
    def move(dx, dy):
        pass

a = Rover()
a.move(1, 1)
```

Exception

```
TypeError: Rover.move() takes
2 positional arguments but 3
were given
```

Mechanism

Supplied: a, 1, 1
Declared: dx, dy
Fix: def move(self, dx, dy)

> 错误定义只写 dx、dy 两个形参。通过实例调用时还会自动传入 a，所以函数实际收到 a、一、一，共三个位置实参，解释器抛 TypeError。修复为 def move(self,dx,dy)。self 是惯用的普通形参名，参数数量和对应关系决定调用能否成立。

Why Did x Not Move?



This candidate changes a **local x**, not the rover's self.x.

Complete Python example

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5      def move(self, dx, dy):
6          x = self.x + dx
7          self.y += dy
8
9  a = Rover(0, 0)
10 a.move(1, 1)
11 print(a.x, a.y)
```

Python output

```
0 1
```

Read the result

x is local to this call.
Fix: self.x = self.x + dx

- 要保留行动结果，就更新实例字段。

37

第 37 页 · Why Did x Not Move?

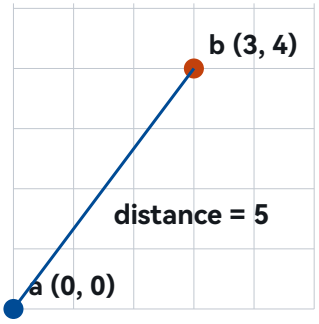
> 检查一个试图重写 move 的候选。横向计算只赋给局部 x，没有存回 self.x，所以实际输出零、一，只有纵坐标移动。把那一行改为 self.x = self.x + dx 才能保留结果。这里仍在完善同一个移动功能；相同规则稍后也用于金币和物品字段。

Next Feature · Measure Distance

Read **self** and **other** without moving either rover.

Add to Rover 添加距离查询

```
def distance_to(self, other):
    dx = self.x - other.x
    dy = self.y - other.y
    return (dx*dx + dy*dy) ** 0.5
```

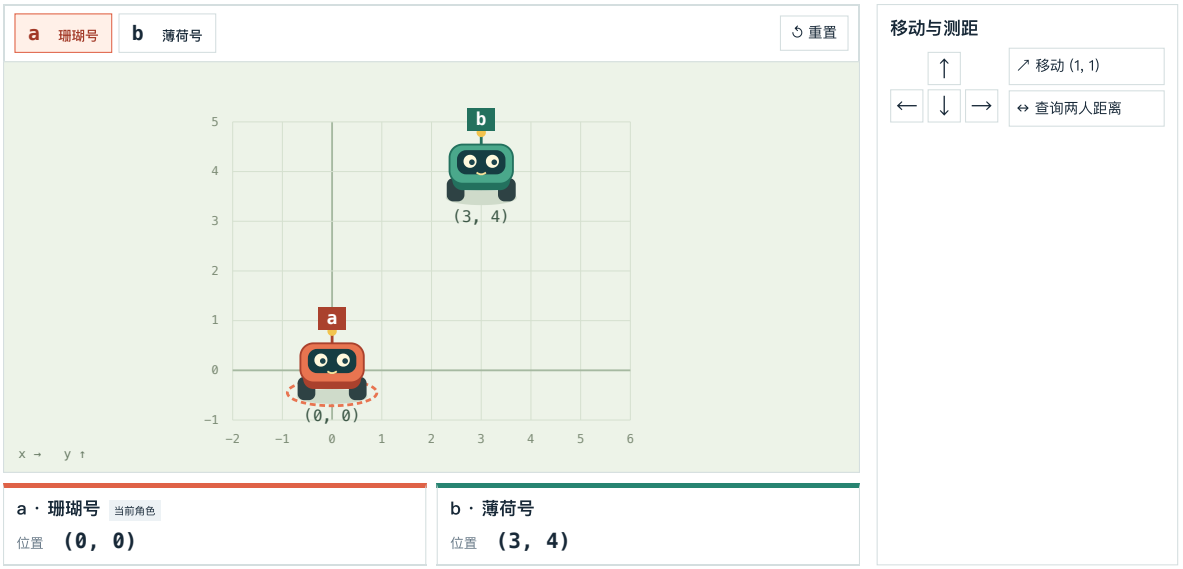


Full program 完整程序

- 本次查询测试：起点 (0, 0) 与 (3, 4)，距离 5.0。

> 位置版还需要查询两个角色的距离。重新用起点零零和三四测试，a.distance_to(b) 中 self 是 a，other 是 b。两坐标差的平方和开根得到五。计算只用局部 dx、dy，不赋值到角色字段，因此它是一个只读查询。

Version 1 · Position



39

第 39 页 · Version 1 · Position

> 第一版已经是一个独立可运行的程序，不只是零散语法。创建两个角色后测得距离五，移动 a 得到一一，b 保持三四。初始化解决统一创建，self 解决当前操作谁，距离方法展示两个对象的只读协作。接下来的新需求是让这个角色带着金币去付款。 选择一个角色并操作，比较两个角色在每一步之后的状态。

Review · Our Position Design

The position algorithm is unchanged; **its organization has changed**.

Procedural · interface excerpt

```
a = {"x": 0, "y": 0}
b = {"x": 3, "y": 4}
move(a, 1, 1)
distance(a, b)
```

Object-oriented · interface excerpt

```
a = Rover(0, 0)
b = Rover(3, 4)
a.move(1, 1)
a.distance_to(b)
```

Design question	Our Rover version
Where is state established?	__init__ creates the instance attributes
Which rover does move change?	The receiver becomes self
Where are related operations defined?	move and distance_to belong to Rover

40

第 40 页 · Review · Our Position Design

> 位置版已经实现，可以回看两种组织方式。移动坐标的计算保持一致；左边把角色记录传给普通函数，右边通过实例访问方法。初始化统一建立坐标，self 明确当前角色，距离查询也放在同一个接口下。字典和函数版本同样能够正确运行。下一步让 Rover 带上金币，继续在这个接口中维护新增规则。

HALFWAY

Break

> 我们已经完成角色的位置、移动和距离查询。下半段继续修改同一个 Rover，增加金币和物品能力，并观察实例状态与方法如何配合。

Instance State and Methods

__init__ creates state; **self** selects the instance to update.

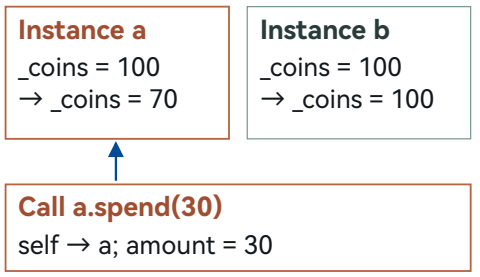
Complete minimal class

```
class Rover:
    def __init__(self, coins=100):
        self._coins = coins
    def spend(self, amount):
        self._coins -= amount
        return True
    def coins_left(self):
        return self._coins

a, b = Rover(), Rover()
print(a.spend(30))
print(a.coins_left(), b.coins_left())
```

Python output

```
True
70 100
```



给同一个角色类型增加金币字段和相关方法。这个最小程序只保留金币成员，完整项目继续保留坐标与背包。每次初始化都给当前实例建立 _coins；调用 a.spend(30) 时，self 指向 a，amount 是这次调用的局部参数。方法结束后，a 的金币字段保留为七十，b 的字段仍为一百。coins_left 通过 self 读取当前实例的金币。对照代码、返回值和两份实例状态，观察初始化、方法与属性的关系。

Reading State and Returning Values

Track **instance state** and **return values** separately.

Method excerpt + caller

```
def coins_left(self):
    return self._coins

a = Rover()
paid = a.spend(30)
coins = a.coins_left()
print(paid, coins)
print(a.coins_left())
```

Python output

```
True 70
70
```

Stored state

a._coins keeps 70 after the call

Method result

paid receives True; coins receives 70

> 接着看对象状态和调用结果分别存在哪里。spend 修改 a 的 _coins，并返回 True，调用者用局部变量 paid 接收；coins_left 读取 _coins，把七十返回给 coins。方法调用结束后，实例字段继续保留；再次读取仍得到七十。返回值可以交给另一个名字，实例属性则通过对象继续访问。本页的 Rover 定义沿用前一页。

Call Another Method Through self

Within a method, use **self.name(...)** to access another instance method.

Complete runnable example

```
class Rover:
    def __init__(self):
        self._coins = 100
    def coins_left(self):
        return self._coins
    def can_pay(self, amount):
        return amount <= self.coins_left()
a = Rover()
print(a.can_pay(30))
```

Python output

```
True
```

Mechanism

self.coins_left() selects the same rover.
Class members require attribute access.

> 方法体内调用同一个对象的另一个方法，写 self.coins_left()。先通过 self 进行属性查找，所得方法绑定到同一对象，再执行调用。这个完整最小示例同样用初始化建立 _coins 字段，再用查询方法读取余额。

A Bare Name Uses Function Scope



The class namespace is **not an enclosing function scope** for a method.

Complete runnable example

```
class Rover:
    def __init__(self):
        self._coins = 100
    def coins_left(self):
        return self._coins
    def can_pay(self, amount):
        return amount <= coins_left()
a = Rover()
print(a.can_pay(30))
```

Exception

```
NameError: name 'coins_left'
is not defined. Did you mean:
'self.coins_left'?
```

Mechanism

coins_left() looks for a plain name.
self.coins_left() looks up an attribute.

> 把 self.coins_left() 改成 coins_left() 后，函数体把 coins_left 当作普通名字，按函数的名字查找规则寻找。类命名空间不充当方法的外层函数作用域，因此这个完整程序抛 NameError。类属性也应通过 self 或类名访问。

Why Call It _coins?

The underscore marks **internal storage**, not an access lock.

Public interface and direct access

```
a.spend(30)
print(a.coins_left())

print(a._coins)  # accessible directly
```

- 公开方法提供接口；下划线约定标记内部存储。

A promise between users and authors

Callers use spend / coins_left.
_coins is internal storage.
Python still allows direct access.

> 使用单下划线把金币存储标作内部实现，让调用者通过 spend 和 coins_left 交互。Python 仍然允许外部读写 a._coins。公开方法说明调用者怎样与对象交互，下划线帮助区分接口与内部存储；这种命名约定保留了直接访问的可能。

Double Underscores Change Names

Within Rover, **__coins is rewritten as _Rover__coins**.

Complete runnable example

```
class Rover:
    def __init__(self):
        self.__coins = 100
a = Rover()
print(vars(a))
print(a._Rover__coins)
print(hasattr(a, "__coins"))
```

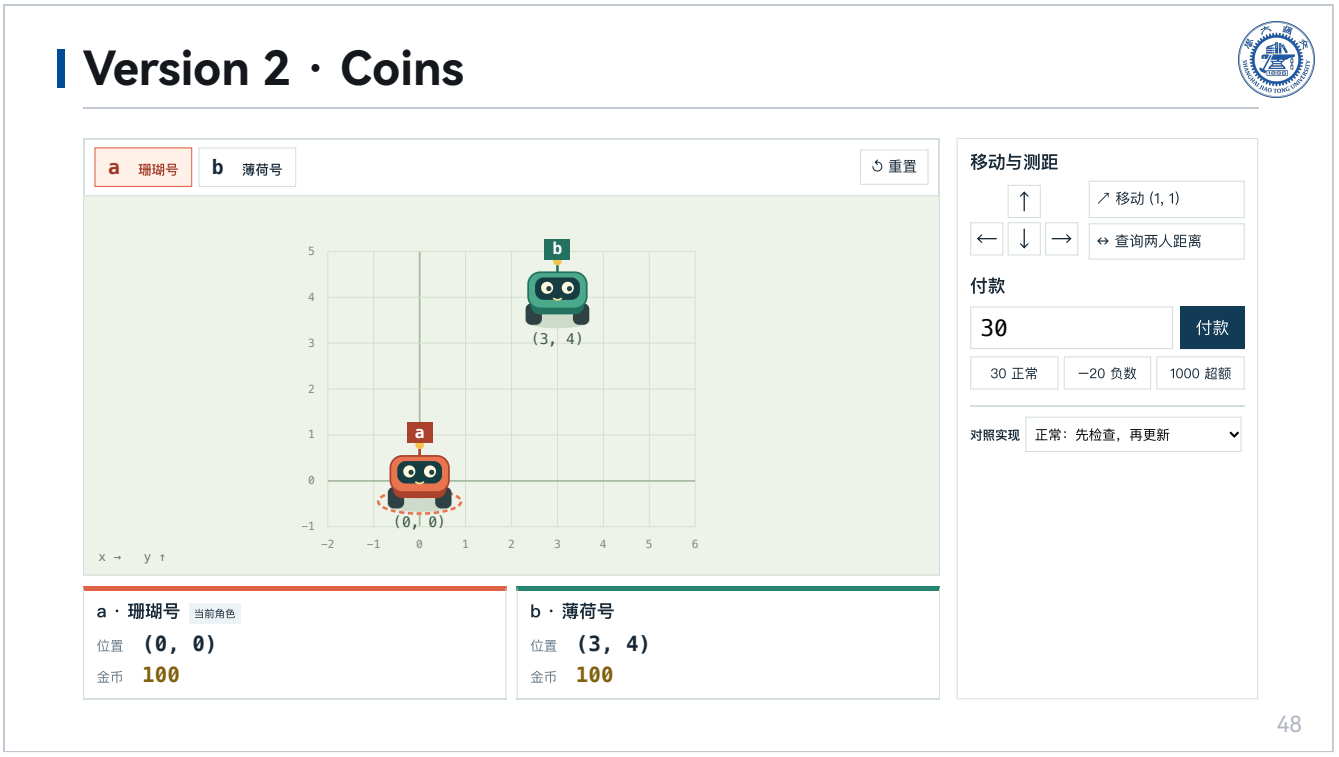
Python output

```
{'_Rover__coins': 100}
100
False
```

Mechanism

__coins → _Rover__coins
_coins → non-public convention
__init__ → special method name

> 在类内部使用双前导下划线且尾部至多一个下划线的标识符会触发名称改写。这里 self.__coins 保存成 _Rover__coins，vars(a) 能看到实际名字，外部也能通过改写后的名字访问。该机制主要减少继承中意外同名的冲突；__init__ 这类双端特殊方法名不按此规则改写。



第 48 页 · Version 2 · Coins

> 第二版在第一版基础上增加金币，没有换成另一个账户例子。现在同一个 Rover 能移动到一一，并付款三十后留下七十金币。我们已经完成开场任务的前两部分，接下来要让角色捡到苹果，并确保另一个角色的背包不跟着变化。 选择一个角色并操作，比较两个角色在每一步之后的状态。

Stage 3 · Let a Pick an Apple

Only **a** picks an apple. Why does **b** have it too?

Complete Python example

```python
class Rover:
    items = []
    def pick(self, item):
        self.items.append(item)

a = Rover()
b = Rover()
a.pick("apple")
print("a:", a.items)
print("b:", b.items)
print(a.items is b.items)
```

Python output

```
a: ['apple']
b: ['apple']
True
```

Read the result

a and b both show apple.
True: a.items and b.items are one list.

- 只让 a 拾取，为什么两个背包都有苹果？

> 现在给角色增加背包。为便于直接运行，本页的最小类只保留 items 和 pick，完整项目中的坐标和金币仍然保留。先创建 a、b，只调用 a.pick("apple")，再分别打印两人的背包。实际输出显示两边都有 apple，身份比较还得到 True：它们访问的是同一个列表。self 仍然选中了 a；接下来追踪 items 存放的位置，比较类体赋值与初始化中的实例属性赋值。

Class Attributes and Instance Attributes

Where the assignment runs determines who owns items.

Class attribute 类属性

One list on **Rover**

Created when the class body runs

```
class Rover:
    items = []
```

a and b find the same **Rover.items**.

Instance attribute 实例属性

A new list on **each rover**

Created on each initialization

```
class Rover:
    def __init__(self):
        self.items = []
```

Each call creates **[]** for its **self**.

- 类体建立共享属性；每次初始化为当前实例建立自己的背包。

> 把前页现象对应到两种赋值位置。左边执行类体时只建立一份 Rover.items，a、b 自己没有 items 时，都到类上找到它，所以修改列表内容会被双方看到。右边每次初始化都会重新求值 []，再把新列表存为当前 self 的实例属性；初始化 a 和 b 得到不同列表。关键是属性绑定在哪里，以及列表创建几次。实例属性也可以由其他方法建立，这里放进 __init__ 是为了让每个新角色一开始就拥有背包。后面先跟踪左侧写法的读取、修改和重新绑定，再回到右侧完成修复。

Instance Assignment Shadows a Class Field

For this ordinary field, **assignment to a.unit creates an instance binding**.

Complete runnable example

```
class Rover:
    unit = "grid"
a, b = Rover(), Rover()
a.unit = "pixel"
print(a.unit, b.unit, Rover.unit)
print(vars(a))
del a.unit
print(a.unit, vars(a))
```

Python output

```
pixel grid grid
{'unit': 'pixel'}
grid {}
```

Mechanism

Read: instance → class
Write a.unit: bind on a
Delete a.unit: reveal the class value

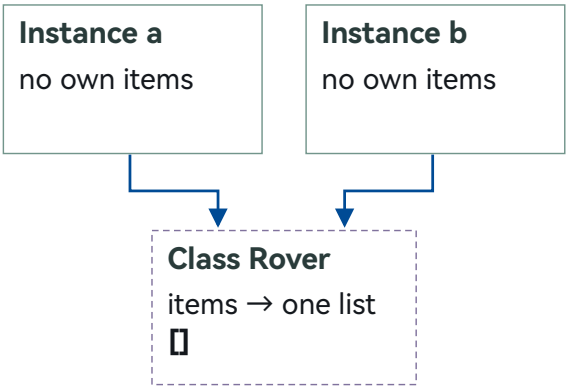
开始时 a、b 都从类读取 unit。a.unit = pixel 后，a 自己有同名字段，b 和 Rover 仍为 grid。删除 a 的字段后，读取又能找到类上的 grid。这里跟踪普通数据字段；下一讲再扩展描述符和继承参与时的属性查找。

Trace the Backpack · 1/3

Neither rover owns items; both read **the class list**.

One statement at a time 一次一步

```
a = Rover()
b = Rover()
```



- 修改列表内容，与重新绑定字段，是两种不同操作。

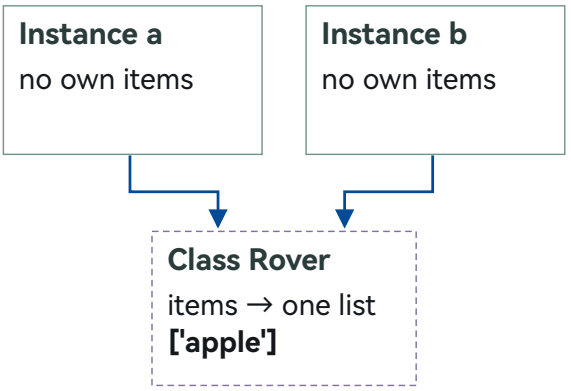
两个角色分别创建，但都没有自己的 items，沿着查找路径找到同一份空列表。 这里跟踪的是加入背包时的错误候选，修复后仍会合并回原来的 Rover。

Trace the Backpack · 2/3

append changes that shared list; **both rovers see apple**.

One statement at a time 一次一步

```
a.items.append("apple")
```



- 修改列表内容，与重新绑定字段，是两种不同操作。

53

第 53 页 · Trace the Backpack · 2/3

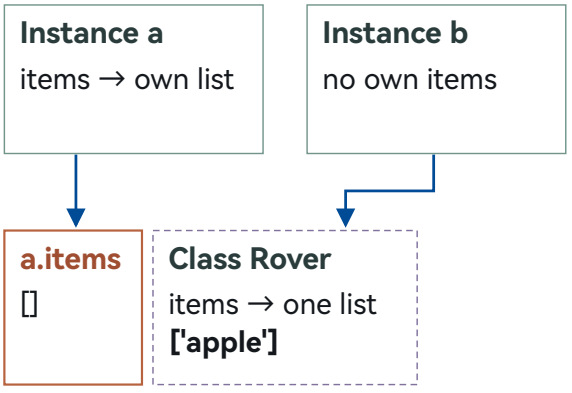
> a 的 append 修改找到的原列表；列表里新增苹果，b 和类属性都能看到。 这里跟踪的是加入背包时的错误候选，修复后仍会合并回原来的 Rover。

Trace the Backpack · 3/3

a.items = [] creates a new binding; **b still reads the old list**.

One statement at a time 一次一步

```
a.items = []
```



- 修改列表内容，与重新绑定字段，是两种不同操作。

54

第 54 页 · Trace the Backpack · 3/3

> 给 a.items 赋新列表，建立 a 自己的同名字段，之后 a 先读自己的空列表，而 b 仍读类上带苹果的列表。 这里跟踪的是加入背包时的错误候选，修复后仍会合并回原来的 Rover。

Give Every Rover Its Own Bag

Create [] during **each initialization**, then keep pick unchanged.

Python excerpt

```
    # Inside the existing __init__:
    self.items = []

def pick(self, item):
    self.items.append(item)
```

Python output

```
['apple'] []
False
```

What changed? 结果

a holds apple; b is empty.
Different lists, same pick method.

Full program 完整程序

- 初始化中保存坐标、金币，并为每个角色建立背包。

> 回到同一个 Rover 的初始化过程，给每个新角色执行一次列表创建。pick 方法不需要更改，self 选中的角色现在有自己的 items。测试显示 a 拾取苹果后 b 仍为空，身份比较为 False。每次初始化创建的新列表保证了背包外层独立。

A Mutable Default Is Reused

The default **[] is created when def runs**, then reused across calls.

Complete runnable example

```
class Rover:
    def __init__(self, items=[]):
        self.items = items
a, b = Rover(), Rover()
a.items.append("apple")
print(a.items, b.items)
print(a.items is b.items)
```

Python output

```
['apple'] ['apple']
True
```

Mechanism

One default list; two stored references.
Use None, then create [] inside the call.

> 把默认参数写成 items=[]，即使赋值位于 __init__，两次省略参数仍会拿到同一个默认列表。self.items = items 只保存引用，a 加苹果后 b 也能看到。每次执行函数体和每次创建默认值是不同的时机，类体执行页的规则在此产生直接后果。下一页用 None 表示未提供，再在初始化中建立新列表。

What If We Supply a Starter Bag?

Two calls can receive one input list. **Copy the outer list** if required.

Inside __init__ 初始化中的输入处理

```
if items is None:
    self.items = []
else:
    self.items = list(items)
```

Test the starter bag 测试初始物品

```
loot = ["map"]
a = Rover(items=loot)
b = Rover(items=loot)
a.pick("apple")
# b.items == ["map"]
# loot == ["map"]
```

Full program 完整程序

- 本例约定背包外层独立；物品是字符串。

> 现在允许地图给角色分配初始物品。如果直接 self.items = items，而两个角色传入同一个 loot，就又会共享。最终版本约定角色拥有自己的外层背包，因此用 list(items) 复制外层列表；未提供时默认 None，再创建空列表。这也避免可变默认列表。这种复制只建立新外层列表，嵌套的可变元素仍需按需求单独分析。

What Does a Shallow Copy Copy?

list(...) copies the outer list; nested objects retain their identities.

Complete runnable example

```
loot = [["map"]]
a = list(loot)
b = list(loot)
a[0].append("key")
print(a is b)
print(a[0] is b[0])
print(b)
```

Python output

```
False
True
[['map', 'key']]
```

Mechanism

Outer lists: different objects
Inner list: one shared object
Our rover items are strings.

> 本讲背包的物品是字符串，复制外层列表已经满足独立添加和删除的需求。这个扩展例子把一个可变列表当作物品：a、b 外层不同，内部第零项仍指向同一列表，因此追加 key 会同时出现在 b 的读取结果里。复制深度应当和数据结构及所有权约定一致。

What Should All Rovers Share?



The game can count **how many rovers were created** in total.

Design excerpt 设计片段

```
class Rover:
    created = 0
    def __init__(self, ...):
        # Initialize the instance fields.
        Rover.created += 1
```

After creating a and b 创建两个角色之后

> 2

Per rover 每个角色

x, y, _coins, items

Per class 整个类

created: one shared counter

- 共享是否正确，先看它属于“一个角色”还是“整个角色类型”。

> 背包共享是错误，因为任务要求每个角色独立；但游戏大厅累计创建过多少个角色，应该是类级信息。最终版本把 created 放在 Rover 类上，每次初始化完成后明确更新 Rover.created。计数在初始化完成时增加。这个数字记录累计成功创建次数；角色离开或引用删除时，计数保持。

Why self.created += 1 Is Wrong Here



For this integer, **+= reads through the class and writes onto the instance**.

Complete runnable example

```
class Rover:
    created = 0
    def __init__(self):
        self.created += 1
a, b = Rover(), Rover()
print(a.created, b.created)
print(Rover.created)
print(vars(a))
```

Python output

```
1 1
0
{'created': 1}
```

Mechanism

Read 0 from Rover → compute 1
Write created=1 onto a
Fix: Rover.created += 1

> 这个候选希望累计创建次数，却写 self.created += 1。a 最初从类读到零，加一得到新整数，再给 a 建立 created 字段；b 也从仍为零的类字段开始。因此两个实例都是一，类仍为零。最终 Rover 实现显式使用 Rover.created += 1 更新共享计数。

Load Spawn Coordinates

Map data arrives as a pair. Add a **named creation entry**.

Python excerpt

```
    @classmethod
    def from_pair(cls, pair):
        return cls(pair[0], pair[1])

b = Rover.from_pair((3, 4))
print(b.x, b.y, b.coins_left())
```

Full program 完整程序

- 新增创建入口，仍复用同一个初始化规则。

Python output

```
3 4 100
```

What changed? 结果

cls receives Rover.
cls(3, 4) uses normal initialization.

> 最后给地图数据一个清晰入口：地图用坐标对表示出生点。Rover.from_pair 接收这份数据，用 cls 创建同一种角色。classmethod 会自动传入类而非实例，惯用名 cls；它再走现有初始化，所以金币和独立背包都不会漏掉。坐标对包含两个坐标是本接口的前提。

Pick the Right Receiver

A helper that needs **no rover state** can be a static method.

Rover helper · used before storing the opening coins

```
    @staticmethod
    def valid_coins(value):
        return type(value) is int and value >= 0
```

Instance method	Class method	Static method
self → one rover a.move(1, 1)	cls → Rover Rover.from_pair(pair)	No automatic receiver Rover.valid_coins(100)

- 先判断需要谁的状态，再选择方法类别。

> 最终版本将初始金币检查提取为 valid_coins 辅助方法。它只看参数，不读某个角色或类的状态，staticmethod 让访问时不自动补 self 或 cls；初始化调用它后才保存金币。比较三种接口：move 需要实例，from_pair 需要类，valid_coins 两者都不需要。纯辅助函数写在类外也正确，这里把它放在角色类中是为了集中这套输入规则。

A Class Method Receives the Class

@classmethod binds cls to the class, through either access form.

Complete runnable example

```
class Rover:
    def __init__(self, x=0):
        self.x = x
    @classmethod
    def at(cls, x):
        return cls(x)
a = Rover()
b = a.at(3)
print(a.at.__self__ is Rover)
print(b.x, b is a)
```

Python output

```
True
3 False
```

Mechanism

Rover.at(3) → function(Rover, 3)
a.at(3) → function(Rover, 3)
cls(3) creates a new instance.

> 这个独立最小例子用 at 表示命名创建入口。Rover.at(3) 和 a.at(3) 都把 Rover 传入 cls；a.at.__self__ 可以验证这一点。cls(3) 重新走初始化，得到横坐标为三的新实例，原来的 a 保留。主程序的 from_pair 使用同一机制，从二元坐标创建角色。

A Static Method Gets Explicit Arguments



@staticmethod adds no automatic receiver, through class or instance.

Complete runnable example

```
class Rover:
    @staticmethod
    def valid_coins(value):
        return (type(value) is int
                and value >= 0)
a = Rover()
print(Rover.valid_coins(100))
print(a.valid_coins(-1))
print(a.valid_coins(True))
```

Python output

```
True
False
False
```

Mechanism

Rover.valid_coins(100) → function(100)
a.valid_coins(-1) → function(-1)
No self or cls is inserted.

> 静态方法使用普通 def 形参，只接收括号内明确给出的值。通过类或实例访问都按这一规则调用，因此这里分别判断一百、负一和 True。使用 type(value) is int 且 value大于等于零，明确排除了布尔值。只读参数的辅助函数也可放在类外；放进 Rover 是为了集中角色金币的规则。



第 65 页 · Complete Mission

> 现在运行开场的完整探险任务。建立 a零零和 b三四，起始距离五；a 移动一一、支付三十、捡苹果，最后为一一、七十金币和一个苹果。b 仍为三四、一百金币和空背包。位置、付款、背包三段代码已经合并到同一个完整 Rover 类，下载 Python 版本可以直接运行；下方游戏显示相同的角色状态变化。 选择一个角色并操作，比较两个角色在每一步之后的状态。

Review · Choosing an Organization

Choose an interface for **related state and operations**.

A procedural design

Records hold x, y, coins and items.
Functions receive those records.
A factory can standardize creation.

Our object-oriented design

Each rover holds its own state.
Methods act through the receiver.
Rover groups fields and related methods.

Situation	A useful starting point
A fixed calculation or data-processing pipeline	Functions that process clearly defined inputs
Many entities with persistent state and related actions	Objects with explicit state and interfaces

- 本讲：类、对象与封装。下一讲：继承与多态。

> 完整任务已经完成。两种组织方式都能产生相同结果。面向过程可以用记录、函数和统一创建入口组织这份程序；本讲选择对象接口，把初始化、实例字段和相关方法集中在 Rover 中。实例之间的共享关系需要根据属性的归属分析。设计时先看哪些状态需要跨调用保留、哪些操作与这些状态紧密相关，再决定函数与类怎样配合。

Review · Class Mechanisms

扫码作答

1 坐标是否存入角色
2 背包的共享与重新绑定
3 绑定方法是否保留接收者

https://taohuang.info/cs1602/zh/quiz/9/

第 67 页 · Review · Class Mechanisms

用刚完成的角色程序检验三个机制。题目使用三个精简的 Rover 程序：第一题查字段保存位置，第二题查同一个列表和重新绑定，第三题查保存方法后接收者是否改变。之后逐题解释。

Review 1 · Where Is x?

A local assignment does **not create self.x**.

Short faulty candidate 精简候选

```
class Rover:
    def __init__(self, x):
        x = x

a = Rover(3)
print(a.x)
```

Full program 完整程序

Python output

```
AttributeError: 'Rover'
object has no attribute 'x'
```

Reasoning 推导

x = x only rebinds a local name.
No self.x was created.

第 68 页 · Review 1 · Where Is x?

局部 x = x 没有建立角色的 self.x。访问 a.x 会抛 AttributeError，正确初始化应把值存入 self.x。

Review 2 · Whose Backpack?

First mutate the shared list; then give a **its own binding**.

Short faulty candidate 精简候选

```
class Rover:
    items = []

a = Rover()
b = Rover()
a.items.append("apple")
a.items = []
print(len(a.items), len(b.items),
      len(Rover.items))
```

Full program 完整程序

Python output

```
0 1 1
```

Reasoning 推导

append changes the class list.
a.items = [] gives only a a new list.

第 69 页 · Review 2 · Whose Backpack?

append 先修改类上的列表，所以 a、b 都看到苹果；随后给 a.items 新空列表，只改变 a 的查找结果，三个长度为零、一、一。

Review 3 · Which Rover Moves?

A stored bound method **keeps its original receiver**.

Bound method example

```
class Rover:
    def __init__(self, x):
        self.x = x
    def move(self, dx):
        self.x += dx

a, b = Rover(0), Rover(10)
step = a.move
a = b
step(2)
print(step.__self__.x, a.x)
```

Full program 完整程序

Python output

```
2 10
```

Reasoning 推导

a = b rebinds the name a.
step still moves the first rover.

第 70 页 · Review 3 · Which Rover Moves?

step 保存的是原先 a 的绑定方法，接收者仍为第一个实例。a = b 只把变量 a 重新绑定到第二个实例。调用 step(2) 更新第一个实例的 x，结果为二；此时 a 指向的第二个实例仍为十。

Class Mechanisms in Rover

Follow **initialization, method binding and attribute ownership**.

`__init__`

Initialization

__init__ runs for each instance
Attributes retain state after the call returns

`self`

Method binding

self selects the receiver
A saved method keeps it
Call parameters are local

`obj.attr`

Attribute ownership

Instances hold their own fields
The class can hold shared data
Lists follow reference identity

- 初始化建立属性 → 方法绑定接收者 → 属性查找与共享关系。

> 用同一个 Rover 回顾类的机制。每次初始化建立当前实例的字段，字段在调用结束后继续保留。普通实例方法通过接收者确定 self，保存的绑定方法也保留接收者；局部参数属于某一次调用。普通属性的查找需要区分实例和类，可变列表是否独立要沿引用检查。坐标、金币和背包分别让这些机制变得可观察。

Use the Same Questions in Lab

Apply the Rover design checks to the **specified lab interfaces**.

Rover 中的问题	对应实验检查
Each rover owns its data	Stack / WordCounter: instance independence
A command updates the intended object	Rectangle.scale: in-place change and return None
Read data held by an instance	Gradebook.average reads that instance's grades

- 实验接口不变；先对照约定，再移用今天的设计方法。

> Rover 是课堂练习；实验继续使用各自题面规定的类型与接口。把同样的设计问题带过去：栈和词计数器是否各自独立，矩形缩放是否修改原对象并返回 None，成绩簿方法是否读取当前实例保存的数据。按既有实验说明提交并完成 AI 使用说明，题目内容和接口保持不变。

Next · More Rover Types

If rover types share actions, **how should we reuse that behavior**?

Inheritance 继承

A fast rover and a cargo rover
Share some actions, change others

Protocols 协议

Let custom objects support familiar Python operations

Modules 模块

Put reusable classes and functions in separate files

> 今天只实现一个 Rover 类型。若探险加入快速角色、载货角色，复制整份代码会带来新的维护问题，下一讲用继承和协议讨论如何复用与替换行为，再用模块组织文件。先有今天清晰的状态与接口，才有可讨论的复用边界。下面七页保留关键用法参考，较长的其他类例子在讲义中独立查阅。

Reference · A Minimal Rover

A complete program: **initialize, update and read** one rover.

Complete Python example

```python
class Rover:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

p = Rover(3, 4)
p.move(1, 0)
print(p.x, p.y)
```

Python output

```
4 4
```

Read the result

默认 Rover() 为 (0, 0)；move 返回 None。

> 复制本页完整程序可以独立运行。每次初始化都建立 x、y，move 接收当前实例并原地修改，最后结果为四四。默认值沿用函数参数规则。更完整的距离查询在讲义，下载包也提供本页代码。

Reference · Three Ways to Share

When state leaks, ask: **where was the list created**?

Source	Cause	Usual choice
Class body: items = []	One list on the class	Create [] on each init call
self.items = items	Two calls receive one list	Copy with list(items) if required
Default: items = []	One default object is reused	Default to None; create on call

Full program 完整程序

- 复制外层列表不会自动复制嵌套对象；有意共享也要写清约定。

第 75 页 · Reference · Three Ways to Share

三种共享路径并不相同。类属性通过查找共享，输入参数保存相同引用，可变默认参数在函数定义时建立后复用。默认参数可用 None 表示未提供，再在调用内创建列表。先决定是否希望共享，再选处理，不要看到共享就盲目复制。

Reference · A Static Helper

`@staticmethod` supplies **neither self nor cls** automatically.

Complete Python example

```python
class Test:
    @staticmethod
    def add(
        u: int,
        v: int,
    ) -> int:
        return u + v

print(Test.add(1, 1))
print(Test().add(3, 4))
```

Python output

```
2
7
```

Read the result

Both calls pass only the explicit arguments.

第 76 页 · Reference · A Static Helper

本页完整程序通过类和实例两种方式访问静态方法，实际分别输出二和七。静态方法适合与类主题紧密相关、但不依赖对象状态的辅助计算。纯工具函数放在类外也完全合理。

Reference · Names and Construction

Names and annotations do **not perform runtime validation**.

Form	Meaning
_name	Nonpublic convention; still accessible
__name	Name mangling in class scope; not security
__init__	Initializes; must return None. Class call returns the instance.
x: float	An annotation; no automatic range check or conversion

讲义中的完整机制与报错示例

> 双下划线主要用于减少名字冲突，例如类内的 __hidden 通常改写为 _类名__hidden。初始化返回非 None 会抛 TypeError。执行 class 语句运行类体，def 建立函数；函数体在调用时才执行。完整程序和真实错误留在讲义，可按需要复现。

Reference · Follow the Bindings

Follow **creation, binding, lookup and sharing** step by step.

Mechanism	What to follow
Construction	Rover(...) creates an instance; __init__ establishes fields
Method binding	a.move combines a function with receiver a
Ordinary attribute lookup	Read the instance, then the class; inspect shadowing
Mutable state	Compare list identities before changing their contents

- 先找名字和属性的归属，再跟踪调用与引用。

> 复习一个类时，沿四个机制追踪：创建时哪些字段属于新实例，通过实例获得的方法保存谁，普通属性从哪里查找，以及两个字段是否引用同一个可变对象。返回值交给调用者的名字，实例字段则继续保存在对象中。完整程序和真实输出可在本讲代码包中复现。

Reference · More Worked Examples

Practice **the same design questions** on different tasks.

Worked example	What to inspect
Triangle	Store sides in attributes; call related methods through self
Dog	Organize fields, external functions and instance methods
HappyNumber	Compute with local current; retain the original number
Vector	Outer-list copy, dimensions; scale mutates the instance

Triangle
Dog · HappyNumber · Vector

> 这些完整例子保留在讲义，按当前问题挑一个读，不需要一次看完。三角形展示属性与方法间调用，Dog 对照外部函数与实例方法，快乐数区分查询工作状态与持久状态，向量比较复制、维度和副作用。每个链接都有完整可运行定义。

Reference · Lab Interfaces

Follow each problem's **return values, effects and failure rules**.

类	Easy-to-miss behavior
Rectangle	scale mutates, returns None; finite positive dimensions
Stack	Empty pop and peek raise IndexError
WordCounter	Exact words; count descending, then word ascending
Temperature	Invalid start → ValueError; rejected update → False
Gradebook	Empty top_student → None; missing average → KeyError

下载本讲可运行示例 ZIP · 查看 Lab 9 完整接口

> 本页用于核对已有实验题面的关键行为，不替代完整范围和评分规则。Gradebook 的并列规则、WordCounter 的词处理约定等均按实验说明。课堂示例的检查策略可复用，但方法名、签名、返回类型不能自行改动。