

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 9

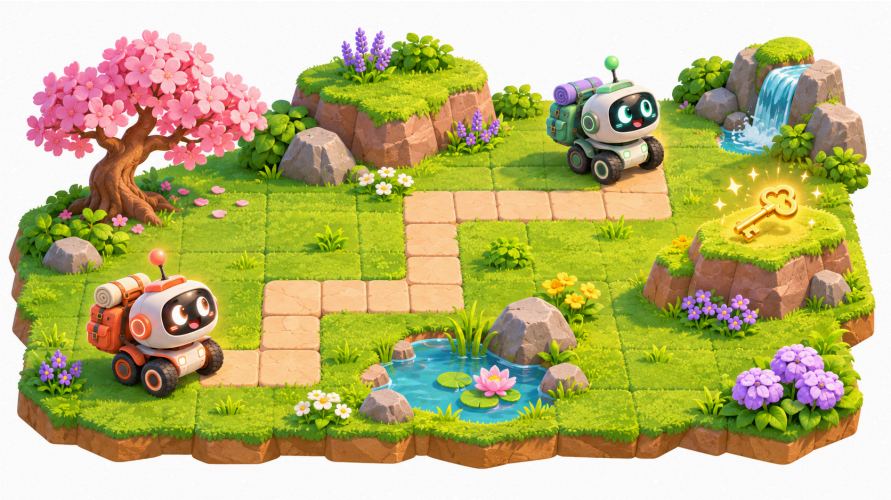
Objects and Encapsulation

Instructor: Tao Huang

Build a Two-Rover Expedition · Fall 2026

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

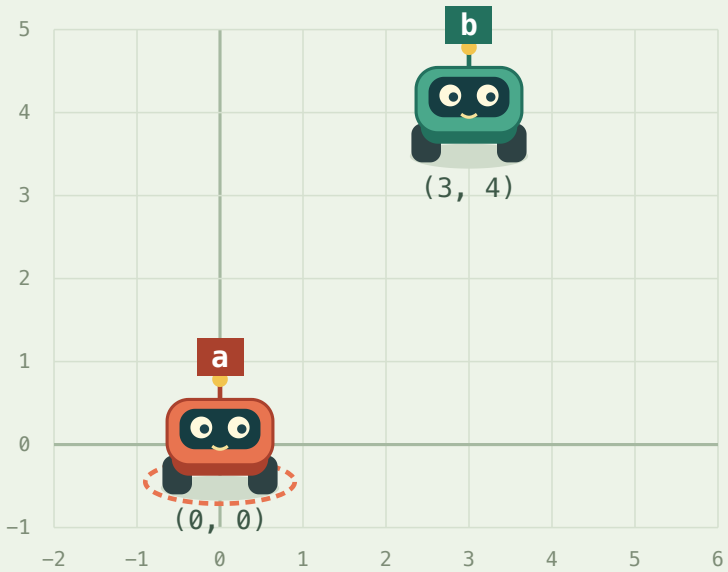
a moves by (1, 1)
a pays 30 coins
a picks an apple

Must stay independent

b keeps its position,
100 coins and an empty bag

Try the Expedition

a 珊瑚号　b 薄荷号　▶ 演示任务　↺ 重置



a · 珊瑚号 当前角色

位置 (0, 0)

金币 100

背包 [] 空背包

b · 薄荷号

位置 (3, 4)

金币 100

背包 [] 空背包

移动与测距

↑　↗ 移动 (1, 1)

←　↓　→　↔ 查询两人距离

付款

30　付款

30 正常　−20 负数　1000 超额

拾取物品

🍎 苹果　🔑 钥匙　🗺️ 地图

对照实现　正常：每个实例自己的列表

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？

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。

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

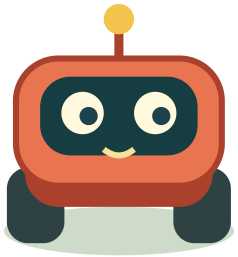
↗ move

spend

pick

update a

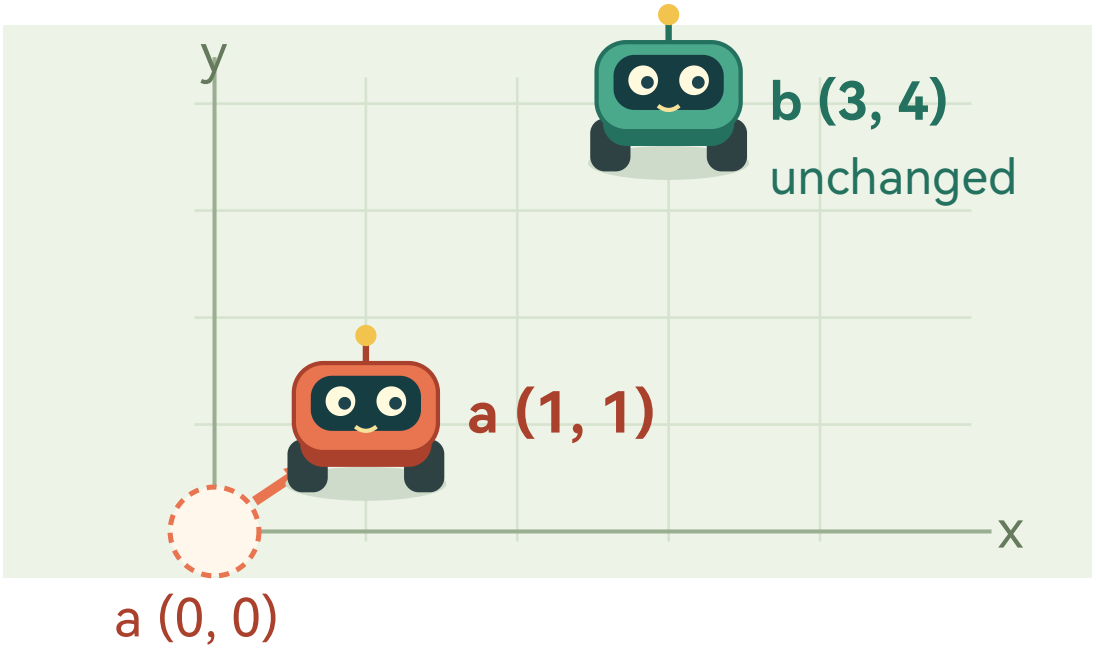
a · data



x = 0, y = 0

100 coins · []

Result · only a moves



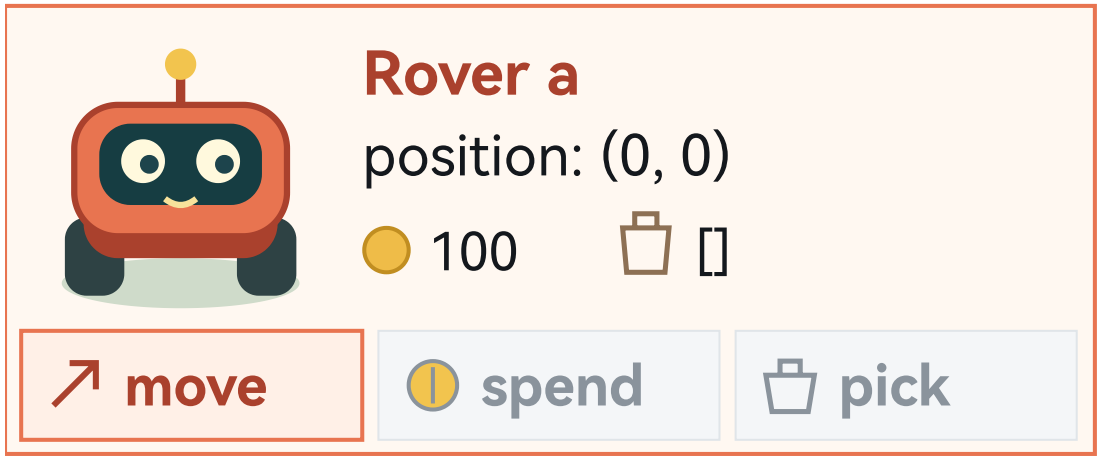

```
move(a, 1, 1)
```

- 调用移动函数，传入需要处理的角色 a。

Object-Oriented Programming

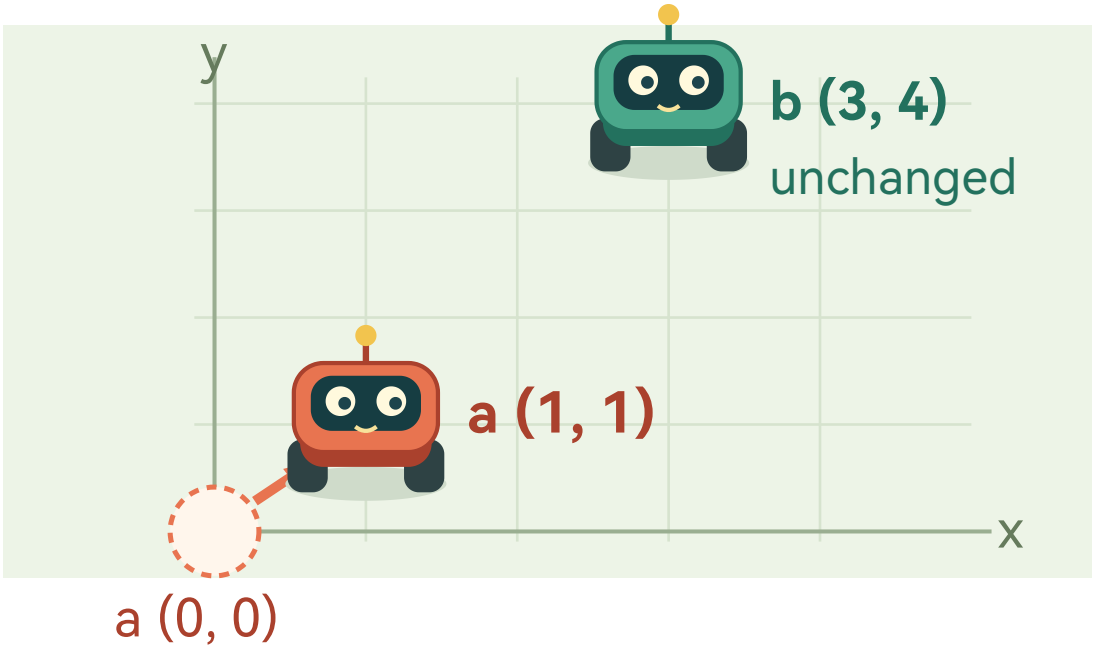
面向对象：围绕**具有状态和行为的对象**组织程序，
通过对象的接口完成操作与协作。

Choose the rover




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

Result · only a moves



- 选中角色 a，调用它提供的移动方法。

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`

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。

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。

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)

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。

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

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.

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。

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'
```

Initialize Rover · 1/5

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

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)
```

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。

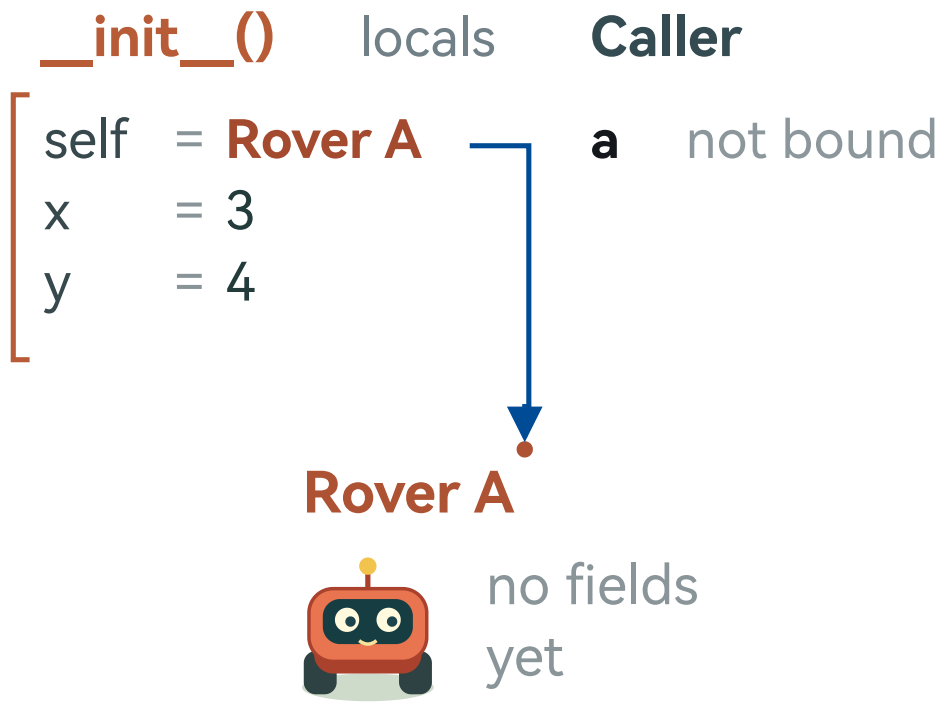
Initialize Rover · 2/5

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。

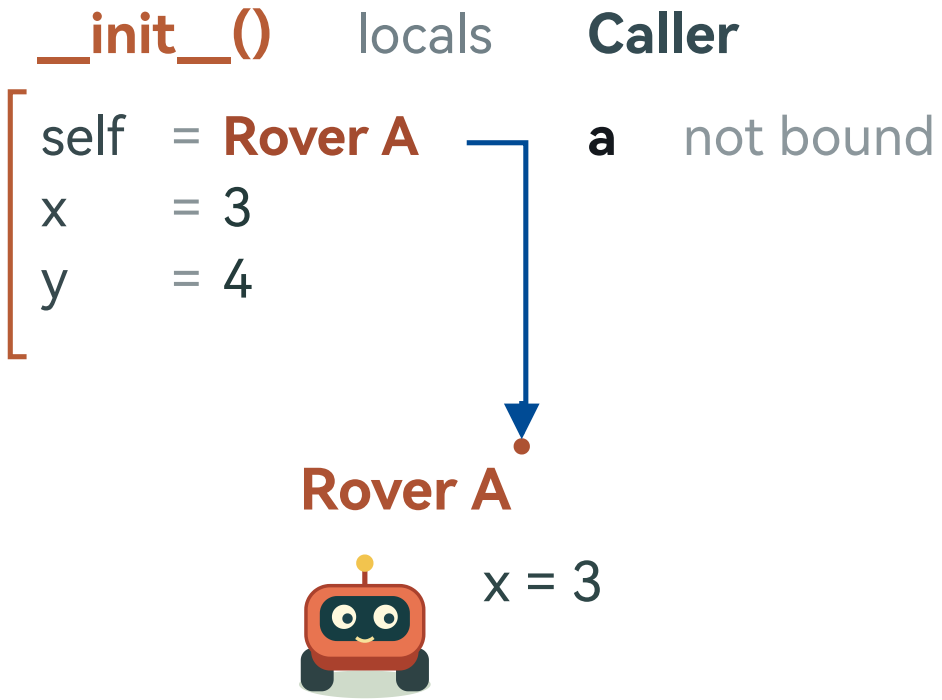
Initialize Rover · 3/5

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。

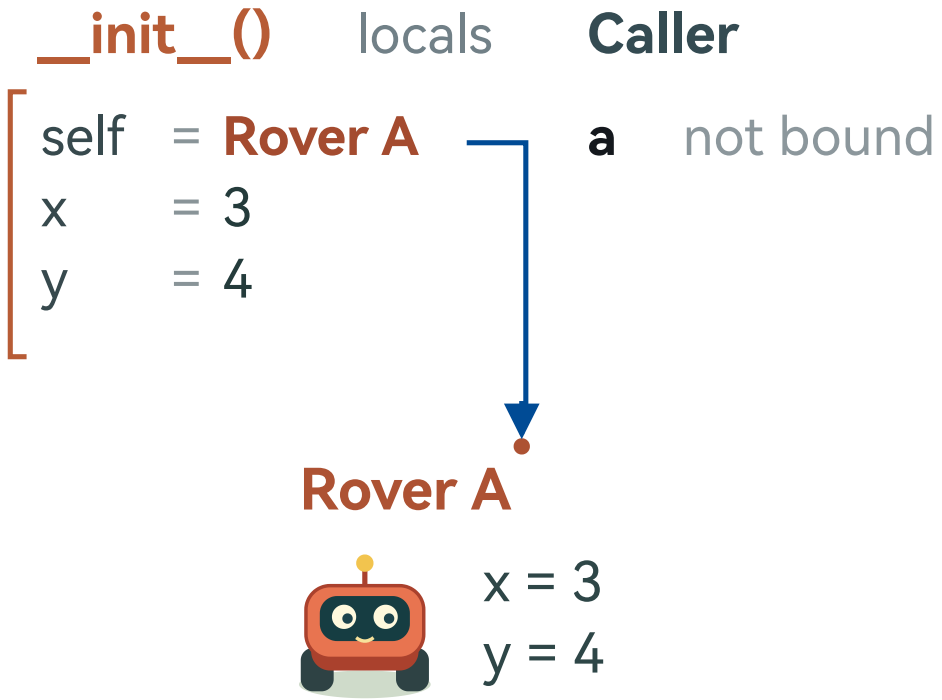
Initialize Rover · 4/5

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。

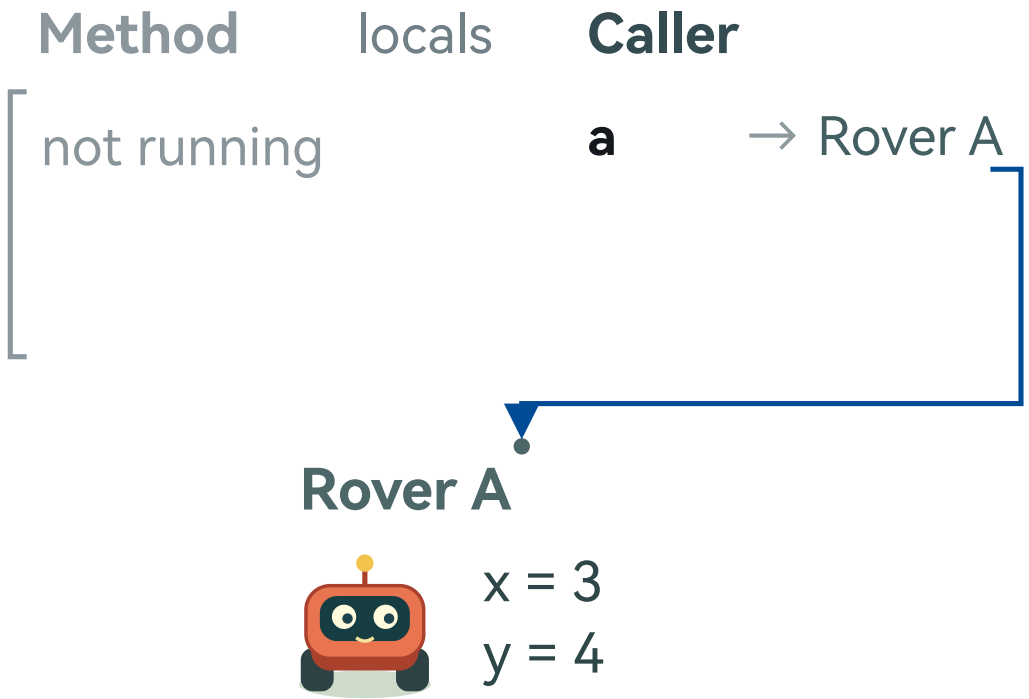
Initialize Rover · 5/5

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。

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

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

Where Do Attributes Live?

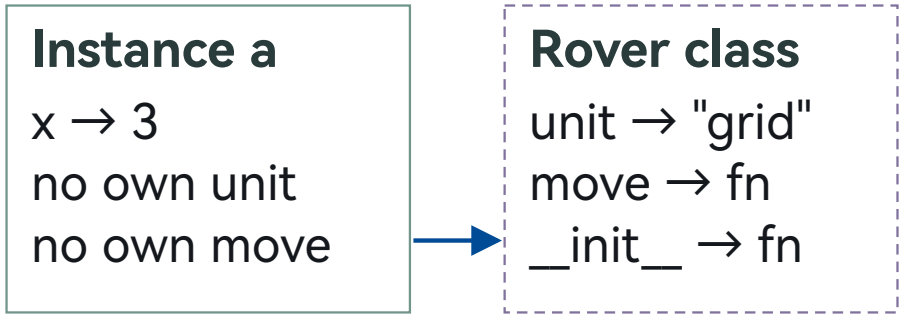
Instance state and the **class namespace** hold different bindings.

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

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。

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
```

- 新问题：一份 move 代码，怎么选中当前角色?

Receiver 接收者

a.move(1, 1)

↓ same receiver and arguments

Rover.move(a, 1, 1)

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)

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)
```

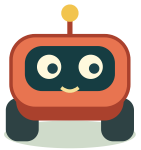
1/5 · next statement to execute

Method locals **Caller**

not running

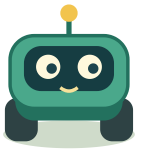
a → Rover A
b → Rover B

Rover A



x = 0
y = 0

Rover B



x = 3
y = 4

- self 选中角色，再更新它的字段。

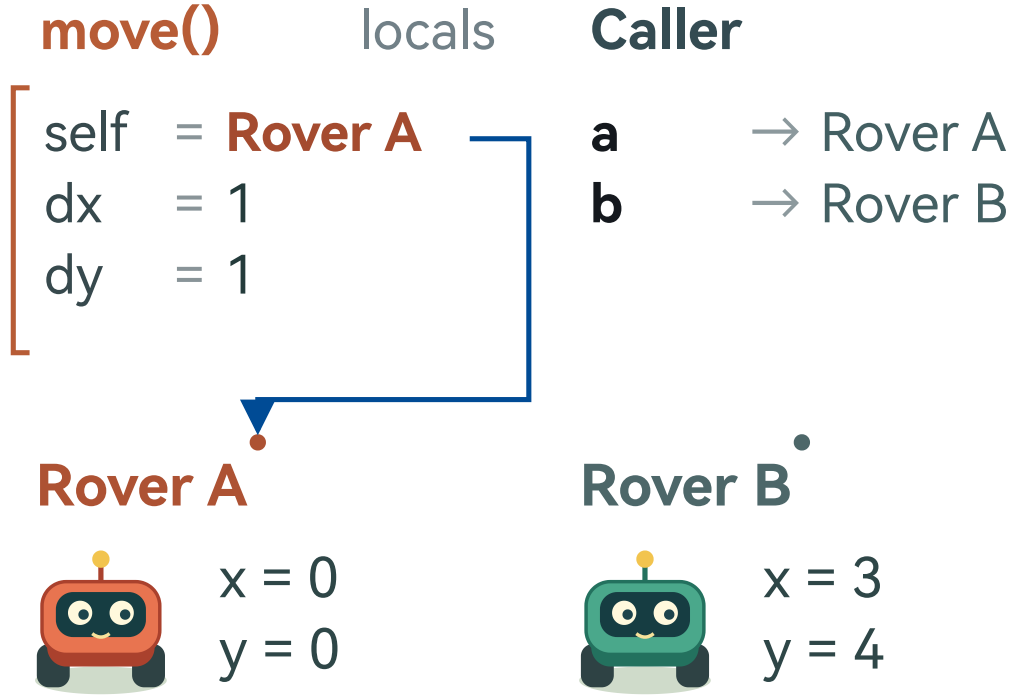
Run a.move · 2/5

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。

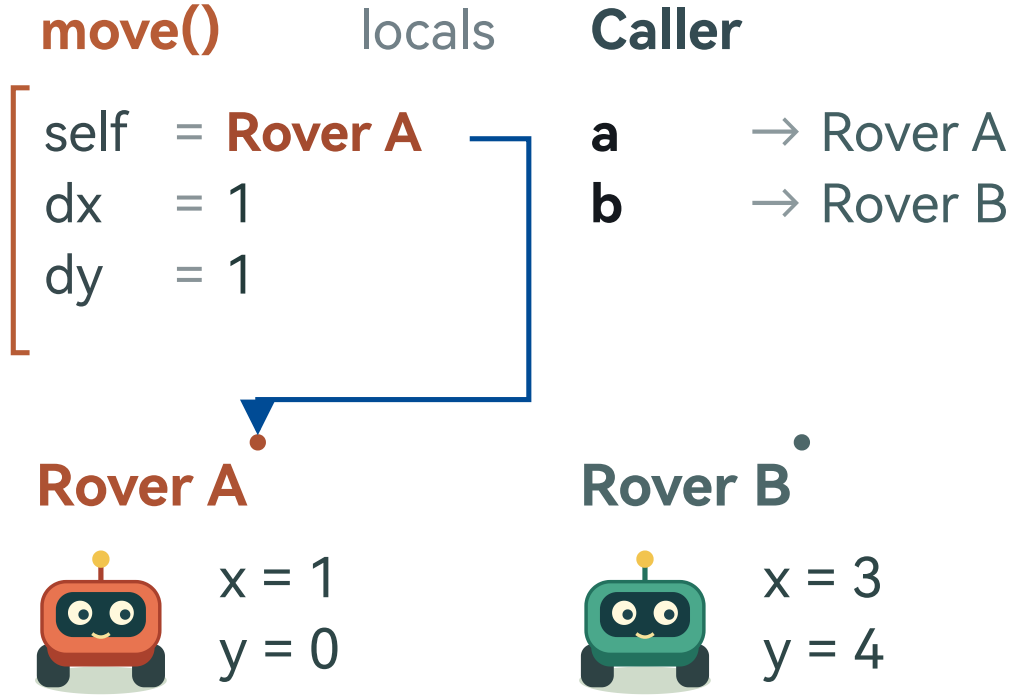
Run a.move · 3/5

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。

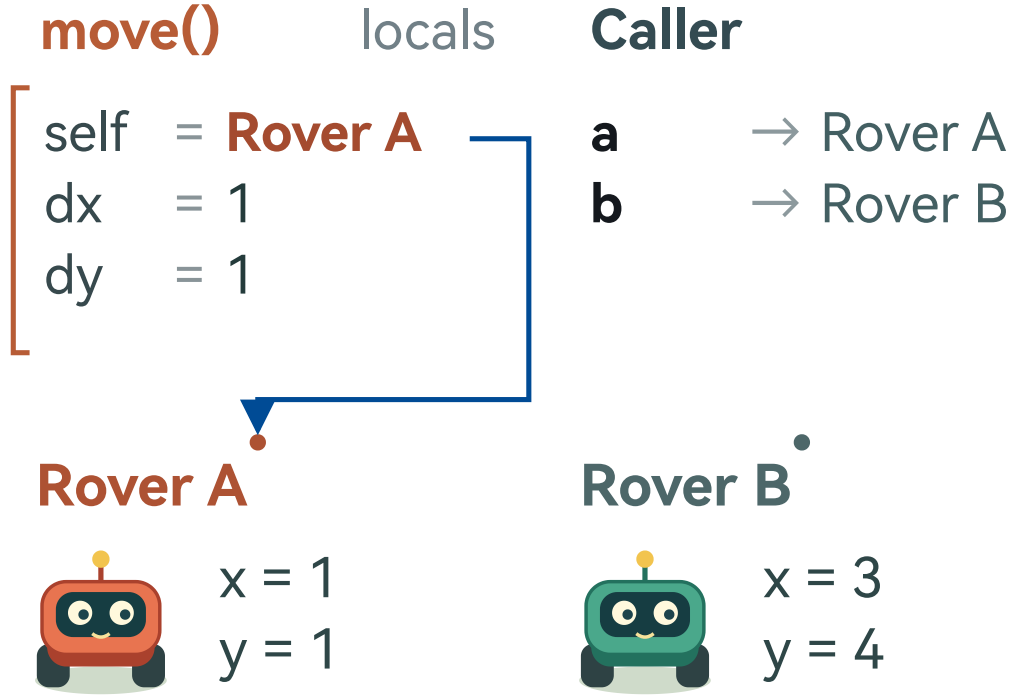
Run a.move · 4/5

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。

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

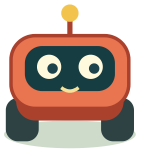
Method locals

not running

Caller

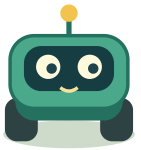
a → Rover A
b → Rover B
result → None

Rover A



x = 1
y = 1

Rover B



x = 3
y = 4

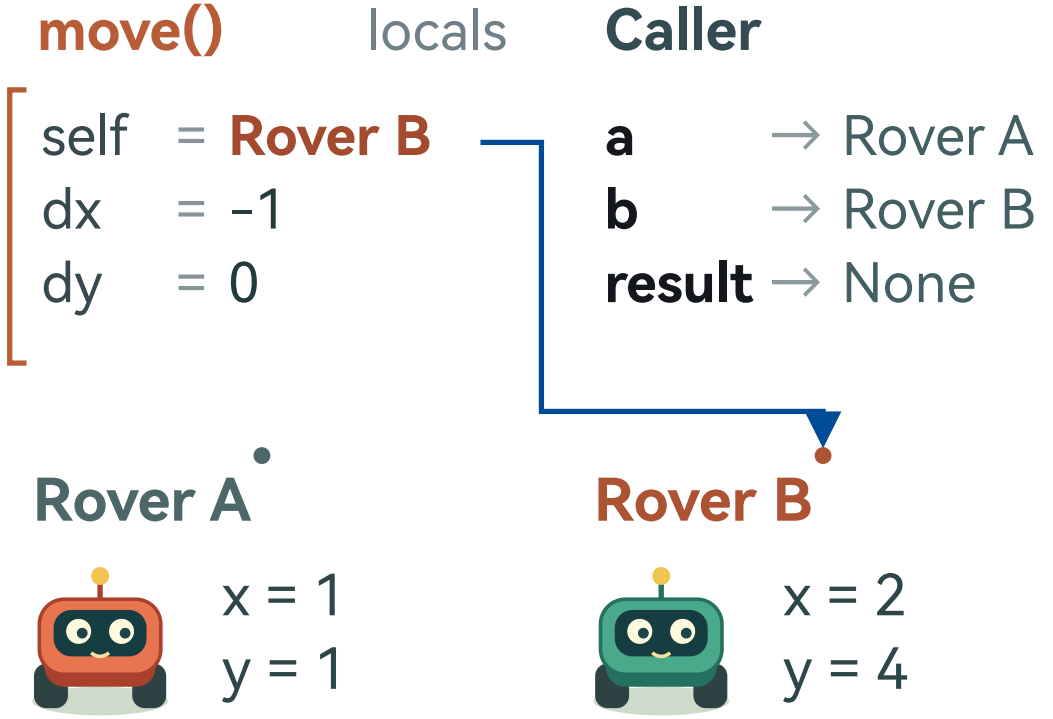
- self 选中角色，再更新它的字段。

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)。

A Bound Method Has Two Parts

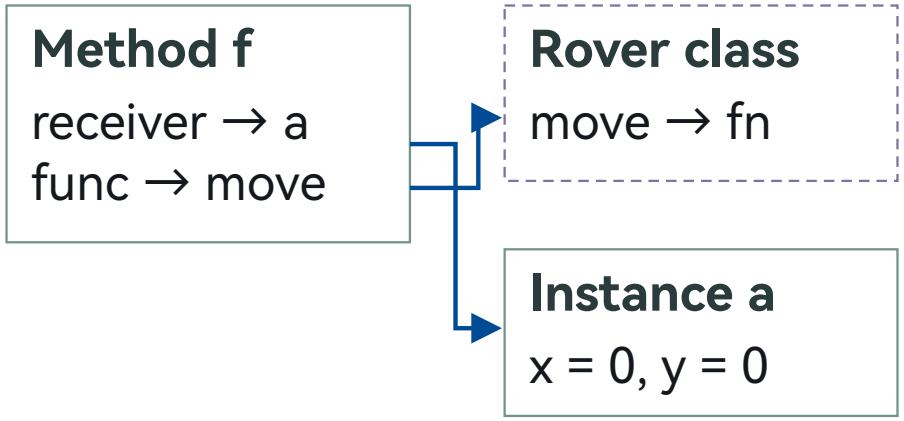
Accessing **a.move** combines a function with the receiver a.

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

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

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)

Why Did x Not Move?

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

Complete Python example

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

a = Rover(0, 0)
a.move(1, 1)
print(a.x, a.y)
```

Python output

```
0 1
```

Read the result

x is local to this call.

Fix: self.x = self.x + dx

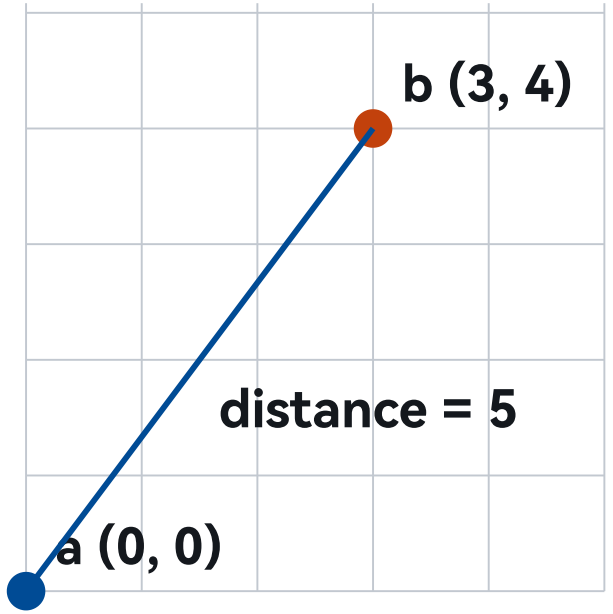
- 要保留行动结果，就更新实例字段。

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
```

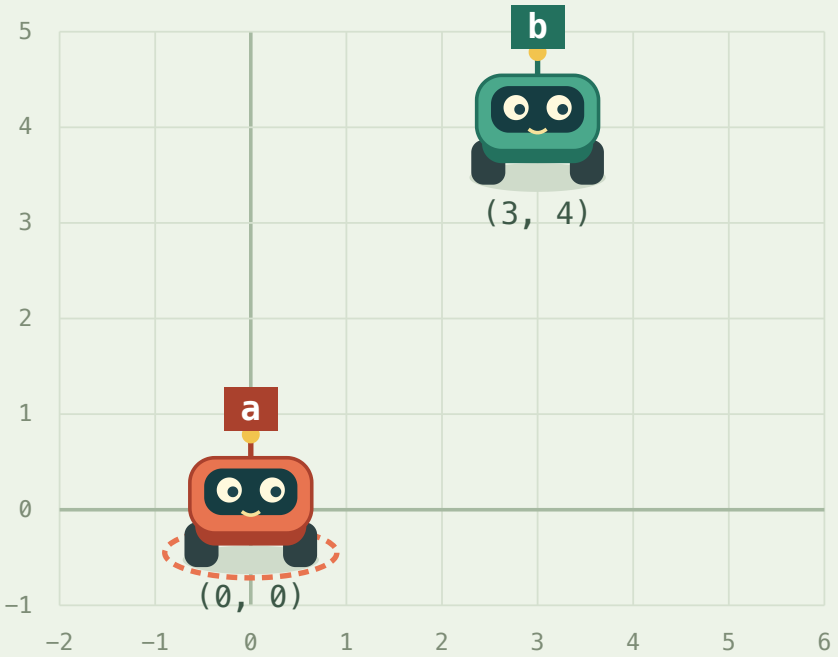


Full program 完整程序

- 本次查询测试：起点 (0, 0) 与 (3, 4)，距离 5.0。

Version 1 · Position

a 珊瑚号　b 薄荷号　重置



移动与测距

↑
← ↓ →

↗ 移动 (1, 1)

↔ 查询两人距离

a · 珊瑚号 当前角色

位置 **(0, 0)**

b · 薄荷号

位置 **(3, 4)**

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

HALFWAY

Break

位置版已完成；接下来给同一个 Rover 加金币和背包。

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
```

Instance a

_coins = 100
→ _coins = 70

Instance b

_coins = 100
→ _coins = 100

Call a.spend(30)

self → a; amount = 30

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

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.

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.

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.

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

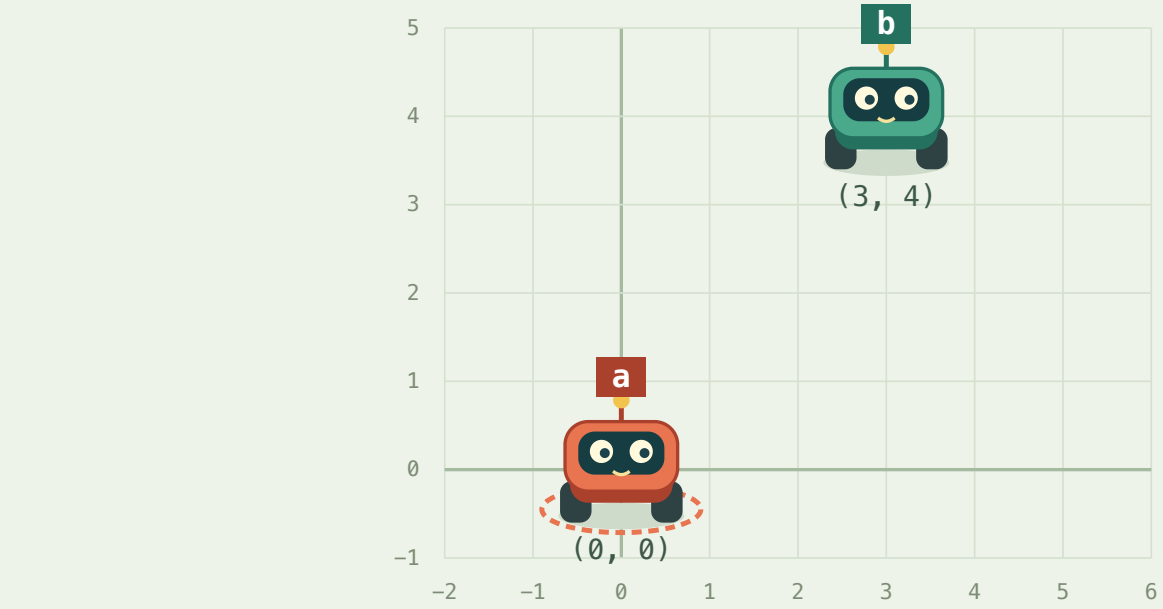
__coins → _Rover__coins

_coins → non-public convention

__init__ → special method name

Version 2 · Coins

a 珊瑚号　b 薄荷号　重置



a · 珊瑚号 当前角色

位置 (0, 0)

金币 100

b · 薄荷号

位置 (3, 4)

金币 100

移动与测距

↑　↗ 移动 (1, 1)

←　↓　→　↔ 查询两人距离

付款

30　付款

30 正常　−20 负数　1000 超额

对照实现　正常：先检查，再更新

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?

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。

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

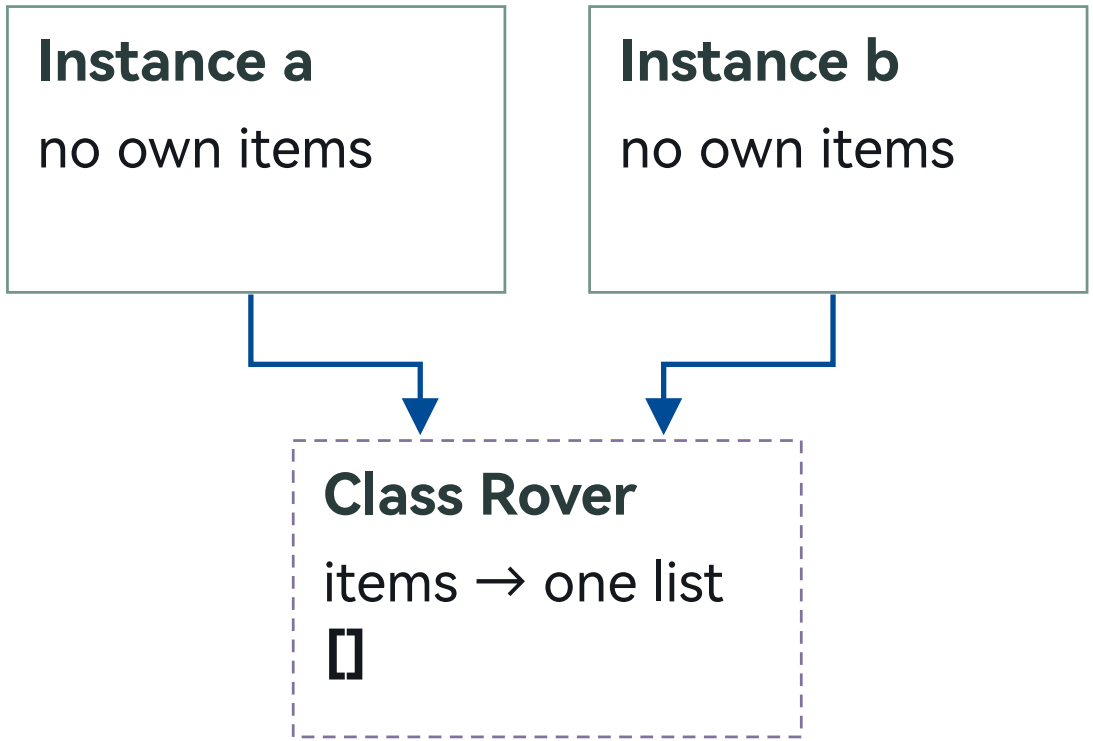
Delete a.unit: reveal the class value

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()
```



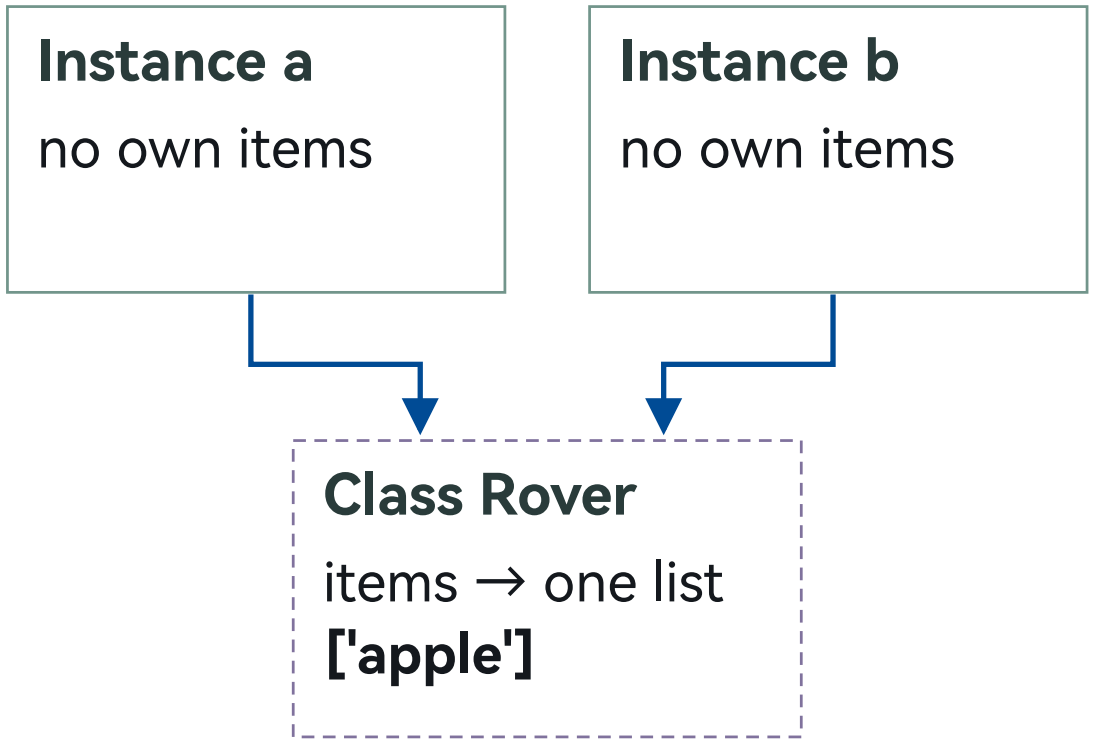
- 修改列表内容，与重新绑定字段，是两种不同操作。

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")
```



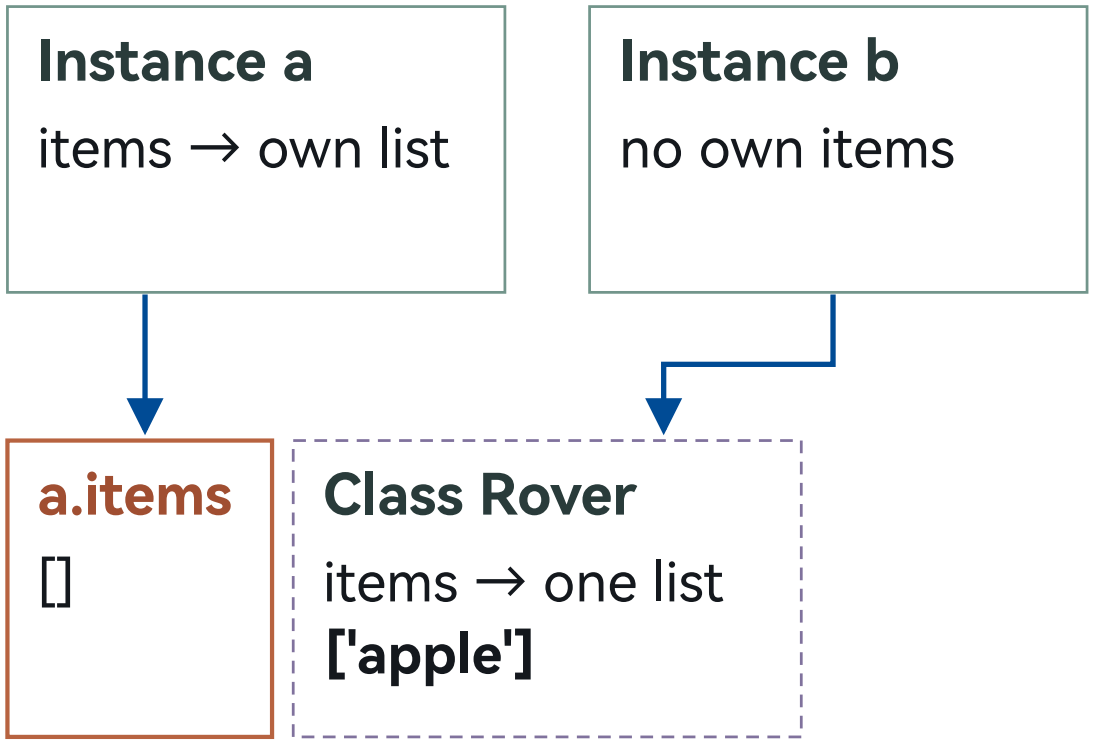
- 修改列表内容，与重新绑定字段，是两种不同操作。

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。

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。

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.

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。

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.

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”。

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

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.

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。

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.

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

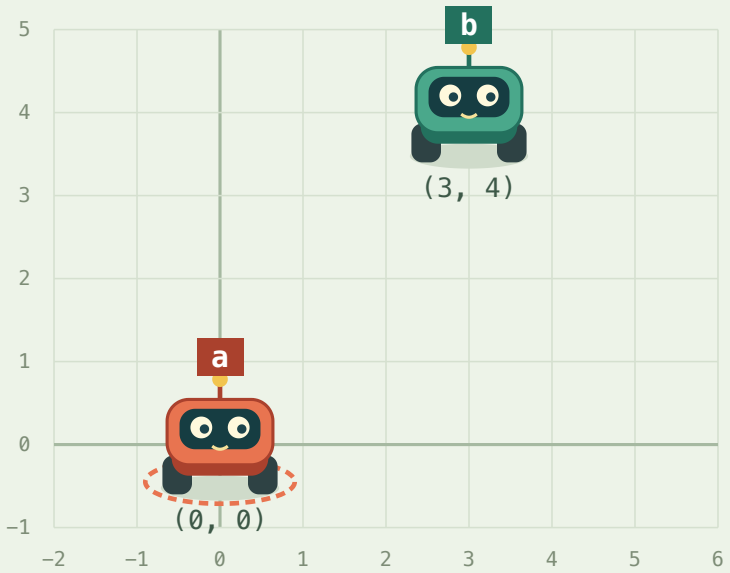
Rover.valid_coins(100) → function(100)

a.valid_coins(-1) → function(-1)

No self or cls is inserted.

Complete Mission

a 珊瑚号 | b 薄荷号 | ▶ 演示任务 | ↺ 重置



a · 珊瑚号 当前角色

位置 (0, 0)

金币 100

背包 [] 空背包

b · 薄荷号

位置 (3, 4)

金币 100

背包 [] 空背包

移动与测距

↑ | ↗ 移动 (1, 1)

← ↓ → | ↔ 查询两人距离

付款

30 | 付款

30 正常 | −20 负数 | 1000 超额

拾取物品

🍎 苹果 | 🔑 钥匙 | 🗺️ 地图

对照实现 正常：每个实例自己的列表

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。

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/

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.

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.

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.

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。

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。

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

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。

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。

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.

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

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。

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

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