

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 10

Classes II: Inheritance, Special Methods and Modules

Instructor: Tao Huang

Part 4: Abstraction and Organization · Fall 2026

The Expedition Continues

Our Rover can move and collect. Now the team needs **different roles**.



Rover

Shared position, movement and backpack

ScoutRover / CargoRover

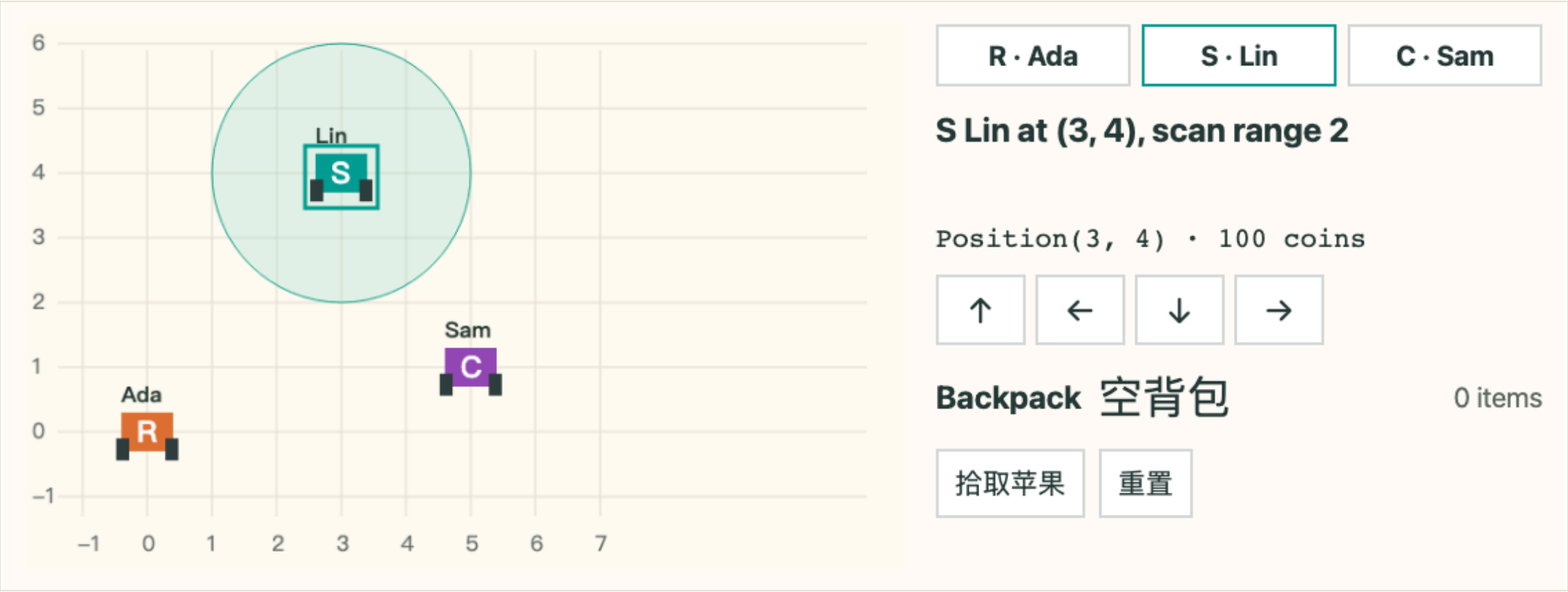
Add role-specific information

Keep the common operations

- 先保留已有能力，再增加角色差异。

Meet the New Team

Move any rover; compare the scout’s range and each backpack.



Copy the Whole Class Again?

Three copied move methods create **three places to maintain**.

Repeated in each class

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

Common

Coordinates · move · pick

Different

Scout: scan_range

Cargo: role-specific description

Can we keep **one** common implementation?

- 新角色需要共同能力，也需要自己的实现。

Today’s Route

Build **one expedition project** in three stages.

1 · Extend

Reuse Rover
Override behavior

→

2 · Integrate

Display objects
Support familiar syntax

→

3 · Organize

Split real files
Control program entry

Download classroom project 示例项目

Inside the ZIP

01_roles/ → 02_protocols/ → 03_expedition/
Run main.py from the selected directory.

Inheritance

A **subclass** can reuse behavior found on a **base class**.

Subclass syntax

```
class ScoutRover(Rover):
    pass
```

继承

子类通过基类关系查找并复用属性和方法。

In this example

ScoutRover → Rover → object
pass adds no new implementation.

- 先认识语法，再观察调用。

An Inherited Method Works

ScoutRover defines no move method; **the inherited move still updates s**.

Complete Python example

```
class Rover:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
class ScoutRover(Rover):
    pass
s = ScoutRover(3, 4)
s.move(1, 0)
print(s.x, s.y)
print(isinstance(s, Rover))
```

Python output

```
4 4
True
```

Read the result

s moves from (3, 4) to (4, 4).

It is also an instance of Rover.

One Instance, Several Places to Find Methods

Method lookup and **instance storage** play different roles.

s · ScoutRover instance

x = 4
y = 4

Fields live on this instance.

Lookup route

ScoutRover
↓ no move here
Rover.move
↓ base of Rover
object

self still refers to **s**.

- 找到哪份函数，与函数操作哪个实例，要分别观察。

Add a Field, Lose the Coordinates?

Defining a new __init__ changes **which initializer is called**.

Complete Python example

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5  class ScoutRover(Rover):
6      def __init__(self, x, y, scan_range):
7          self.scan_range = scan_range
8  s = ScoutRover(3, 4, 2)
9  print(s.x, s.y, s.scan_range)
```

Python output

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

Read the result

The subclass initializer runs.

No assignment creates x or y.

Initialize a Scout · 1/6

ScoutRover.__init__ starts with **one new ScoutRover instance**.

Python · execution

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5  class ScoutRover(Rover):
6      def __init__(self, x, y, scan_range):
▶ 7          super().__init__(x, y)
8          self.scan_range = scan_range
9  s = ScoutRover(3, 4, 2)
10 print(s.x, s.y, s.scan_range)
```

1/6 · next statement

ScoutRover.__init__ · running

self → **ScoutRover**

x = 3, y = 4, scan_range = 2

One ScoutRover instance

(no fields yet)

Initialize a Scout · 2/6

super().__init__ enters Rover; **self is still the same scout**.

Python · execution

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5  class ScoutRover(Rover):
6      def __init__(self, x, y, scan_range):
7          super().__init__(x, y)
8          self.scan_range = scan_range
9  s = ScoutRover(3, 4, 2)
10 print(s.x, s.y, s.scan_range)
```

Rover.__init__ · running

self → **ScoutRover**

x = 3, y = 4

ScoutRover.__init__ · waiting

self → **ScoutRover**

x = 3, y = 4, scan_range = 2

One ScoutRover instance

(no fields yet)

2/6 · next statement

Initialize a Scout · 3/6

self.x has been stored; **self.y is the next assignment**.

Python · execution

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
▶ 4          self.y = y
5  class ScoutRover(Rover):
6      def __init__(self, x, y, scan_range):
7          super().__init__(x, y)
8          self.scan_range = scan_range
9  s = ScoutRover(3, 4, 2)
10 print(s.x, s.y, s.scan_range)
```

Rover.__init__ · running

self → **ScoutRover**

x = 3, y = 4

ScoutRover.__init__ · waiting

self → **ScoutRover**

x = 3, y = 4, scan_range = 2

One ScoutRover instance

```
x = 3
```

3/6 · next statement

Initialize a Scout · 4/6

Rover.__init__ returns; **x and y now belong to the scout**.

Python · execution

```
 1 class Rover:
 2     def __init__(self, x, y):
 3         self.x = x
 4         self.y = y
 5 class ScoutRover(Rover):
 6     def __init__(self, x, y, scan_range):
 7         super().__init__(x, y)
 8         self.scan_range = scan_range
 9 s = ScoutRover(3, 4, 2)
10 print(s.x, s.y, s.scan_range)
```

Rover.__init__ · running

self → **ScoutRover**

x = 3, y = 4

ScoutRover.__init__ · waiting

self → **ScoutRover**

x = 3, y = 4, scan_range = 2

One ScoutRover instance

```
x = 3
y = 4
```

4/6 · return event

Initialize a Scout · 5/6

Back in ScoutRover.__init__, **scan_range is the next field**.

Python · execution

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5  class ScoutRover(Rover):
6      def __init__(self, x, y, scan_range):
7          super().__init__(x, y)
▶ 8          self.scan_range = scan_range
9  s = ScoutRover(3, 4, 2)
10 print(s.x, s.y, s.scan_range)
```

5/6 · next statement

ScoutRover.__init__ · running

self → **ScoutRover**

x = 3, y = 4, scan_range = 2

One ScoutRover instance

```
x = 3
y = 4
```

Initialize a Scout · 6/6

Initialization finishes with **x, y and scan_range on one instance**.

Python · execution

```
1  class Rover:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5  class ScoutRover(Rover):
6      def __init__(self, x, y, scan_range):
7          super().__init__(x, y)
▶ 8          self.scan_range = scan_range
9  s = ScoutRover(3, 4, 2)
10 print(s.x, s.y, s.scan_range)
```

6/6 · return event

ScoutRover.__init__ · running

self → **ScoutRover**

x = 3, y = 4, scan_range = 2

One ScoutRover instance

```
x = 3
y = 4
scan_range = 2
```

Reuse and Extend a Method

super() continues along **the MRO**; the receiver stays the same scout.

Complete Python example

```python
class Rover:
    def describe(self):
        return "at (3, 4)"
class ScoutRover(Rover):
    def describe(self):
        base = super().describe()
        return base + ", scan range 2"
s = ScoutRover()
print(s.describe())
```

Python output

```
at (3, 4), scan
range 2
```

Read the result

MRO 方法查找顺序:
ScoutRover → Rover → object
super starts after ScoutRover.

Override a Method

A subclass can provide **another implementation with the same name**.

Complete Python example

```
1  class Rover:
2      def symbol(self):
3          return "R"
4      def report(self):
5          return self.symbol()
6  class ScoutRover(Rover):
7      def symbol(self):
8          return "S"
9  s = ScoutRover()
10 print(s.report())
```

Python output

```
S
```

Read the result

s.report() returns S.

Next: follow the call that produced it.

Follow the Receiver · 1/4

Rover.report is running with **self → ScoutRover**.

Python · execution

```
1  class Rover:
2      def symbol(self):
3          return "R"
4      def report(self):
5          return self.symbol()
6  class ScoutRover(Rover):
7      def symbol(self):
8          return "S"
9  s = ScoutRover()
10 print(s.report())
```

1/4 · next statement

Rover.report · running

self → **ScoutRover**

One ScoutRover instance

(no fields yet)

Follow the Receiver · 2/4

self.symbol() selects **ScoutRover.symbol**.

Python · execution

```
 1 class Rover:
 2     def symbol(self):
 3         return "R"
 4     def report(self):
 5         return self.symbol()
 6 class ScoutRover(Rover):
 7     def symbol(self):
 8         return "S"
 9 s = ScoutRover()
10 print(s.report())
```

2/4 · next statement

ScoutRover.symbol · running

self → **ScoutRover**

Rover.report · waiting

self → **ScoutRover**

One ScoutRover instance

(no fields yet)

Follow the Receiver · 3/4

ScoutRover.symbol returns **"S"** to the waiting report.

Python · execution

```
1  class Rover:
2      def symbol(self):
3          return "R"
4      def report(self):
5          return self.symbol()
6  class ScoutRover(Rover):
7      def symbol(self):
8          return "S"
9  s = ScoutRover()
10 print(s.report())
```

3/4 · return event

ScoutRover.symbol · running

self → **ScoutRover**

Rover.report · waiting

self → **ScoutRover**

One ScoutRover instance

```
(no fields yet)
```

return **'S'**

Follow the Receiver · 4/4

Rover.report returns **the same result "S"** to its caller.

Python · execution

```
1  class Rover:
2      def symbol(self):
3          return "R"
4      def report(self):
5          return self.symbol()
6  class ScoutRover(Rover):
7      def symbol(self):
8          return "S"
9  s = ScoutRover()
10 print(s.report())
```

4/4 · return event

Rover.report · running

self → **ScoutRover**

One ScoutRover instance

```
(no fields yet)
```

return **'S'**

One Loop, Different Implementations

The same report() call can use **different symbol implementations**.

Complete Python example

```python
class Rover:
    def symbol(self):
        return "R"
    def report(self):
        return self.symbol()
class ScoutRover(Rover):
    def symbol(self):
        return "S"
class CargoRover(Rover):
    def symbol(self):
        return "C"
team = [Rover(), ScoutRover(), CargoRover()]
for rover in team:
    print(rover.report())
```

Python output

```
R
S
C
```

Read the result

R → ordinary rover

S → scout · C → cargo

A Shared Interface Can Cross Class Families

Compatible behavior can work **without a shared custom base class**.

Complete Python example

```
1  class ScoutRover:
2      def report(self):
3          return "S: ready"
4  class Beacon:
5      def report(self):
6          return "B: online"
7  for device in [ScoutRover(), Beacon()]:
8      print(device.report())
9  print(issubclass(Beacon, ScoutRover))
```

Python output

```
S: ready
B: online
False
```

Read the result

Both devices provide report().

Beacon is not a ScoutRover subclass.

Inheritance or Composition?

A scout **is a rover**; a rover **has a backpack**.

Inheritance 继承

ScoutRover → Rover

Reuse a rover's supported behavior

A kind of rover

```
class ScoutRover(Rover):
    ...
```

Composition 组合

Rover → its Backpack object

Store and use another object

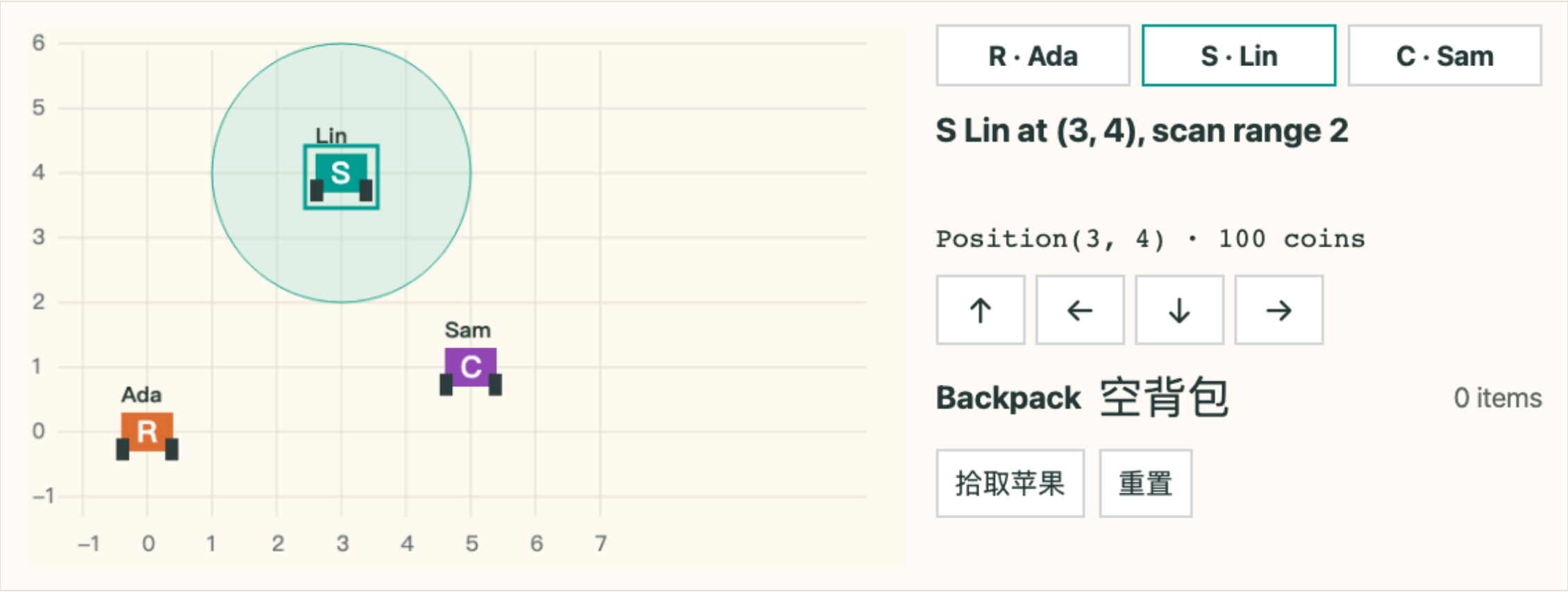
A rover owns a backpack

```
class Rover:
    def __init__(self):
        self.bag = Backpack()
```

- 按支持的行为选择继承，按拥有关系组织组件。

Stage 1 · Roles Work Together

Common movement; role-specific descriptions; **separate bags**.



Make Objects Readable

print(s), repr(s) and a list display use **representation methods**.

Complete Python example

```python
class Rover:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return "Rover " + self.name
    def __repr__(self):
        return f"Rover({self.name!r})"
s = Rover("Ada")
print(s)
print(repr(s))
print([s])
```

Python output

```
Rover Ada
Rover('Ada')
[Rover('Ada')]
```

Read the result

print(s): readable status
repr(s), [s]: informative object description

Can Our Position Use +?

Defining coordinates alone does not define **what addition means**.

Complete Python example

```
1  class Position:
2      def __init__(self, x, y):
3          self.x, self.y = x, y
4  p = Position(3, 4)
5  print(p + Position(1, 0))
```

Python output

```
TypeError:
unsupported operand
type(s) for +:
'Position' and
'Position'
```

Read the result

The expression needs an addition implementation.

Define Position Addition

__add__ creates **a new Position** while preserving p.

Complete Python example

```python
class Position:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        if not isinstance(other, Position):
            return NotImplemented
        x = self.x + other.x
        y = self.y + other.y
        return Position(x, y)
    def __repr__(self):
        return f"Position({self.x}, {self.y})"
p = Position(3, 4)
q = p + Position(1, 0)
print(p, q, p is q)
```

Python output

```
Position(3, 4)
Position(4, 4) False
```

Read the result

p stays (3, 4).

q becomes (4, 4); p is q is False.

Say Which Operands Are Supported

Return **NotImplemented** to let operator dispatch continue.

Complete Python example

```python
class Position:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        if not isinstance(other, Position):
            return NotImplemented
        x = self.x + other.x
        y = self.y + other.y
        return Position(x, y)
    def __repr__(self):
        return f"Position({self.x}, {self.y})"
p = Position(3, 4)
print(p.__add__("east") is NotImplemented)
print(p + "east")
```

Python output

```
True
TypeError:
unsupported operand
type(s) for +:
'Position' and 'str'
```

Read the result

A returned sentinel and a raised exception have different jobs.

Equal Coordinates, Different Objects

__eq__ compares coordinates; **is checks object identity**.

Complete Python example

```
1  class Position:
2      def __init__(self, x, y):
3          self.x, self.y = x, y
4      def __eq__(self, other):
5          if not isinstance(other, Position):
6              return NotImplemented
7          return (self.x == other.x
8                  and self.y == other.y)
9  a = Position(3, 4)
10 b = Position(3, 4)
11 c = a
12 print(a == b, a is b, a is c)
```

Python output

```
True False True
```

Read the result

True: equal coordinates

False: distinct instances

True: c aliases a

Let a Backpack Behave Like a Container

len, indexing and membership each use **a supported protocol**.

Complete Python example

```python
class Backpack:
    def __init__(self, items):
        self._items = list(items)
    def __len__(self):
        return len(self._items)
    def __getitem__(self, index):
        return self._items[index]
    def __contains__(self, item):
        return item in self._items
bag = Backpack(["map", "apple"])
print(len(bag), bag[0])
print("apple" in bag)
```

Python output

```
2 map
True
```

Read the result

2 items · first item is map

apple is in the bag

A Familiar Expression, a Defined Operation

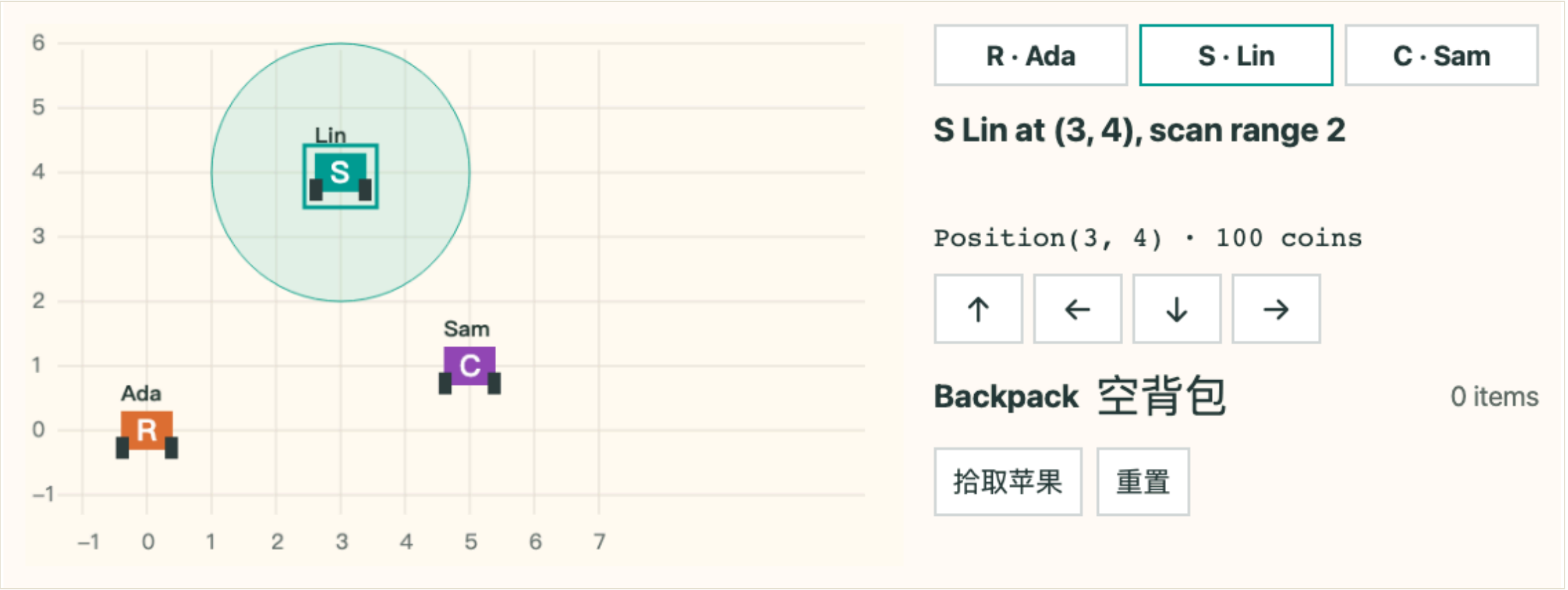
A special method connects **Python syntax** to **our implementation**.

Expression	Method	Meaning in our project
print(rover)	__str__	Readable role status
p + offset	__add__	New position; preserve inputs
a == b	__eq__	Compare coordinates
len(bag), bag[i], item in bag	Container protocols	Read backpack contents

- 先明确操作的含义，再实现Python约定的方法。

Stage 2 · Familiar Operations

Read role descriptions, position values and backpack contents.



The Project Needs More Than One File

Separate **reusable definitions** from **the mission that runs them**.



Definitions

Rover family · Position · Backpack

Entry point

Create team → perform mission → display results

Start with Two Real Files

Save both files in one directory; run **python main.py**.

rover.py

```
class Rover:
    def symbol(self):
        return "R"
    def report(self):
        return self.symbol()
class ScoutRover(Rover):
    def symbol(self):
        return "S"
```

main.py

```
from rover import ScoutRover
s = ScoutRover()
print(s.report())
```

Python 3.13 · actual files

> S

Files: split two/ · run python main.py

A Module Provides a Namespace

import rover binds a **module object**, whose attributes include Rover.

main.py · alternative import

```
import rover
s = rover.ScoutRover()
print(s.report())
```

main namespace

rover → module object

s → ScoutRover instance

rover module namespace

Rover → class object

ScoutRover → class object

Two Import Forms, Two Local Bindings

Both forms reach the class; **the names available in main differ**.

Bind the module name

```
import rover
s = rover.ScoutRover()
```

Bind the selected class name

```
from rover import ScoutRover
s = ScoutRover()
```

Read the origin

Use explicit imports. An alias changes the local access name.

Importing Starts the Mission?

Top-level statements run when the module is initialized.

rover.py

```python
print("mission demo")
class Rover:
    pass
```

main.py

```python
import rover
print("main")
```

Python 3.13 · actual files

```
mission demo
main
```

Files: import_side_effect/ · run python main.py

Keep the Demo Behind an Entry Guard

__name__ records **how this module is being executed**.

rover.py

```
print("loading rover")
class Rover:
    pass
if __name__ == "__main__":
    print("mission demo")
```

Run python rover.py

loading rover
mission demo

Run python main.py

loading rover
main

- 可复用定义留在顶层，任务入口放在条件块里。

Follow an Import · 1/6

main reaches its **first import rover**.

main.py

Python · execution

```
▶ 1 import rover
  2 import rover
  3 print("main")
```

rover.py

Python · execution

```
  1 print("loading rover")
  2 class Rover:
  3     pass
  4 if __name__ == "__main__":
  5     print("mission demo")
```

Now executing

main.py:1

__name__

'__main__'

Visible names

(no Rover / rover binding yet)

rover in sys.modules:
False

Follow an Import · 2/6

Python starts rover.py with **__name__ = "rover"**.

main.py

Python · execution

```
1 import rover
2 import rover
3 print("main")
```

rover.py

Python · execution

```
▶ 1 print("loading rover")
  2 class Rover:
  3     pass
  4 if __name__ == "__main__":
  5     print("mission demo")
```

Now executing

rover.py:1

__name__

'rover'

Visible names

(no Rover / rover binding yet)

rover in sys.modules:
True

Follow an Import · 3/6

The class statement will establish **Rover in the module namespace**.

main.py

Python · execution

```
1 import rover
2 import rover
3 print("main")
```

rover.py

Python · execution

```
1 print("loading rover")
▶ 2 class Rover:
3     pass
4 if __name__ == "__main__":
5     print("mission demo")
```

Now executing

rover.py:2

__name__

'rover'

Visible names

(no Rover / rover binding yet)

rover in sys.modules:
True

Follow an Import · 4/6

The entry condition is false; **the demo print is skipped**.

main.py

Python · execution

```
1 import rover
2 import rover
3 print("main")
```

rover.py

Python · execution

```
1 print("loading rover")
2 class Rover:
3     pass
▶ 4 if __name__ == "__main__":
5     print("mission demo")
```

Now executing

rover.py:4

__name__

'rover'

Visible names

Rover

rover in sys.modules:
True

Follow an Import · 5/6

main reaches its second import; **the cached module is reused**.

main.py

Python · execution

```
1 import rover
2 import rover
3 print("main")
```

rover.py

Python · execution

```
1 print("loading rover")
2 class Rover:
3     pass
4 if __name__ == "__main__":
5     print("mission demo")
```

Now executing

main.py:2

__name__

'__main__'

Visible names

rover

rover in sys.modules:
True

Follow an Import · 6/6

main continues; the final output is **loading rover, then main**.

main.py

Python · execution

```
1 import rover
2 import rover
3 print("main")
```

rover.py

Python · execution

```
1 print("loading rover")
2 class Rover:
3     pass
4 if __name__ == "__main__":
5     print("mission demo")
```

Now executing

main.py:3

__name__

'__main__'

Visible names

rover

rover in sys.modules:
True

from Imports Keep Object References

Rebinding a module attribute and mutating a list have **different effects**.

settings.py

```
count = 1
items = []
```

main.py

```
import settings
from settings import count, items
settings.count = 2
settings.items.append("apple")
print(count, settings.count, items)
```

Python 3.13 · actual files

```
1 2 ['apple']
```

Files: bindings/ · run python main.py

Follow Names Across Two Namespaces

Imported names keep references; **they are not live aliases of module fields**.

main namespace

count → 1
items → shared list

settings namespace

count → 2
items → shared list

One list object

['apple']

Same mechanism as L9

Assignment changes a binding.
append changes a list.

From Files to a Package

A regular package groups related modules under **one import path**.

```
expedition/
    __init__.py
    __main__.py
    rover.py
    geometry.py
    backpack.py
```

Inside expedition/__main__.py

```python
from .rover import ScoutRover
```

Run from the parent directory

python -m expedition

Read the complete explanation 讲义

Which random Did Python Find?

A same-named local file can change **which module gets imported**.

random.py

```
label = "local file"
```

main.py

```
import random
from pathlib import Path
print(Path(random.__file__).name)
print(random.randint(1, 3))
```

Python 3.13 · actual files

```
random.py
AttributeError: module 'random' has no attribute 'randint' (consider
renaming '<project>/random.py' since it has the same name as the standard
library module named 'random' and prevents importing that standard library
module)
```

Files: shadowing/ · run python main.py

Inspect the Actual Environment

Check **the interpreter**, **the module source** and **the matching pip**.

Run in the selected interpreter

```
import sys
import random
print(sys.executable)
print(random.__file__)
```

Terminal

python -m pip --version

VS Code

Select the intended Python interpreter.
Compare it with sys.executable.

- 先确认运行的是哪一个Python、导入的是哪一个模块。

Keep Project Dependencies Together

A virtual environment gives the project **its own dependency environment**.

Create and inspect

```
python -m venv .venv
# Activate it for your platform.
python -m pip --version
```

Typical workflow

Create .venv
Select its interpreter
Install the packages this project needs

Our classroom project

Uses only the standard library.
No extra package installation is required.

Read the complete explanation 讲义

Two Caches, Two Different Jobs

sys.modules keeps **module objects**; .pyc can cache **compiled code**.

Inside this process

import rover → initialize module
import rover again → reuse module object

On disk

__pycache__/…pyc
May reduce recompilation in a fresh process

CPython execution

Source → compile → execute

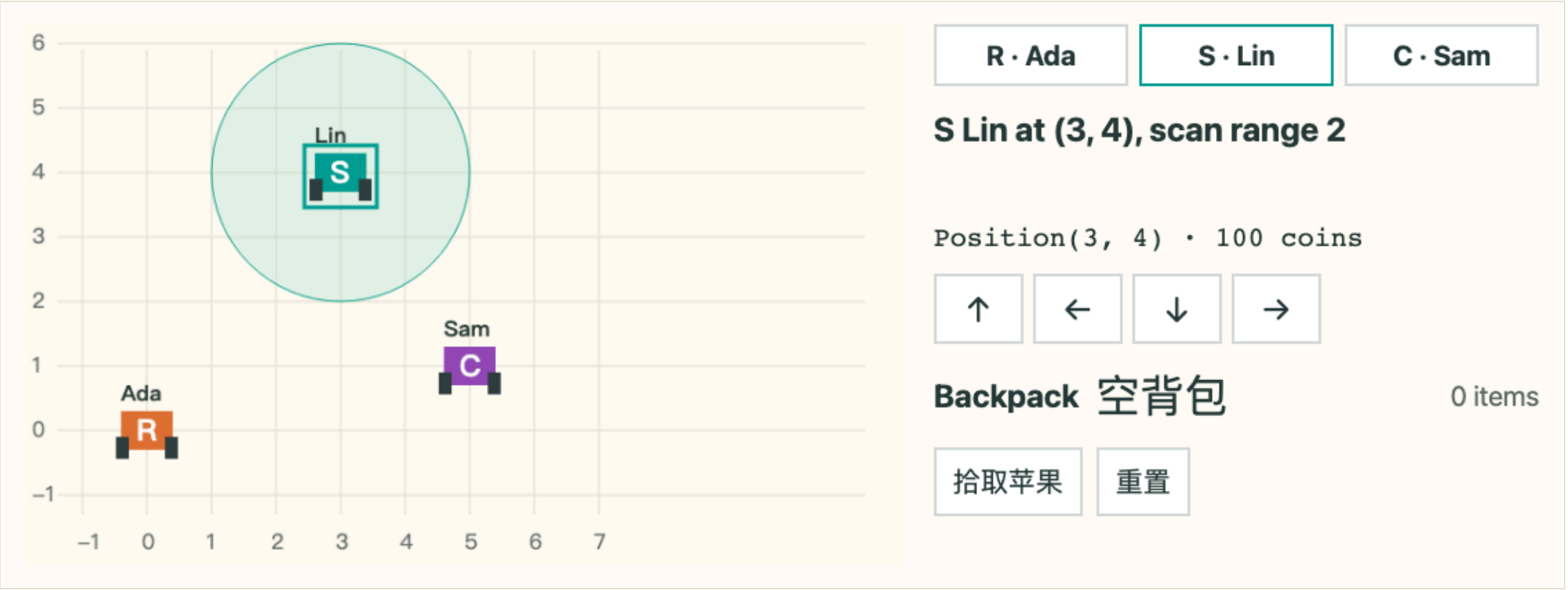
A fresh process

Still executes module initialization.
Disk cache does not skip top-level work.

- 同进程复用模块对象，新进程仍初始化模块。

The Complete Expedition

The multi-file project keeps **the same observable role behavior**.



Review · This Expedition



扫码作答

1 继承调用　　2 相等与身份

3 导入与入口

https://taohuang.info/cs1602/zh/quiz/10/

Review 1 · Which symbol Runs?

A base method keeps **the actual receiver**.

Complete Python example

```python
class Rover:
    def symbol(self):
        return "R"
    def report(self):
        return self.symbol()
class ScoutRover(Rover):
    def symbol(self):
        return "S"
s = ScoutRover()
print(Rover.report(s))
```

Python output

```
S
```

Read the result

Answer: S. Even Rover.report(s) keeps s as the receiver.

Review 2 · Equal or Identical?

Coordinate equality and object identity answer **different questions**.

Complete Python example

```python
class Position:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __eq__(self, other):
        if not isinstance(other, Position):
            return NotImplemented
        return (self.x == other.x
                and self.y == other.y)
a = Position(3, 4)
b = Position(3, 4)
c = a
b.x = 9
print(a == b, a is b, a is c)
```

Python output

```
False False True
```

Read the result

Answer: False False True.

Review 3 · What Runs on Import?

The second import reuses the module; **the demo guard stays false**.

rover.py

```python
print("load")
if __name__ == "__main__":
    print("demo")
```

main.py

```python
import rover
import rover
print("main")
```

Python output

```
load
main
```

What We Built and What Makes It Work

One project now has **related classes, supported operations and reusable modules**.

Extend

Inherited methods
Same receiver
Coordinated initialization

→

Integrate

Representation
Position operations
Backpack access

→

Organize

Module namespaces
Import initialization
Guarded entry

Download classroom project 示例项目

Transfer the Mechanisms to Lab 10

Practise the mechanisms, then build a **three-file shape report**.

Lab task	Mechanism to transfer
Shapes / Rover	Polymorphism · super() · instance state
Money / Deck	Supported operands · equality · container protocols
Shape report	Three real files · quiet imports · a guarded entry

Solo assignment 2

Counting primes: implement the assigned methods and record evidence.
See the course schedule and submission page for the deadline.

Lab 10 · Solo assignment 2

Next · Work, Failures and Evidence

A working program raises three further questions.

How much work?

What changes as the input grows?

How do we check correctness?

Use explicit cases and compare outcomes.

How should failures behave?

Which errors should callers handle?

Lecture 11

Complexity · exceptions · testing

Reference · super and the MRO

super() follows the MRO **after the current class**.

Complete cooperative example

```python
class Root:
    def run(self):
        return ["Root"]
class Left(Root):
    def run(self):
        return ["Left"] + super().run()
class Right(Root):
    def run(self):
        return ["Right"] + super().run()
class Team(Left, Right):
    pass
print(Team().run())
```

Python output

```
['Left', 'Right', 'Root']
```

Actual lookup order

Team → Left → Right → Root → object

Read the complete explanation 讲义

Reference · Special Methods

Use a method only after defining **its result, supported inputs and side effects**.

Operation	Method
Display	__str__ / __repr__
Value comparison	__eq__ / __lt__
Arithmetic	__add__ / __radd__
Length / indexing / membership	__len__ / __getitem__ / __contains__

NotImplemented versus NotImplementedError

Return the sentinel for unsupported operands.
Raise the exception when the method's implementation deliberately fails.

Read the complete explanation 讲义

Reference · Operators and Aliases

__iadd__ may mutate; **ordinary + can create a new object**.

Complete Python example

```python
class Value:
    def __init__(self, n):
        self.n = n
    def __add__(self, other):
        return Value(self.n + other.n)
    def __iadd__(self, other):
        self.n += other.n
        return self
a = Value(1)
b = a
a += Value(2)
print(a.n, b.n, a is b)
a = a + Value(3)
print(a.n, b.n, a is b)
```

Python output

```
3 3 True
6 3 False
```

Read the result

First: one object changes.

Then: only a is rebound.

For scalar × vector, see __rmul__ in the notes.

Reference · Package Layout and Launch

Run a package in its **package context**.

```
04_package/
  expedition/
    __init__.py
    __main__.py
    rover.py
    geometry.py
    backpack.py
```

expedition/__main__.py

```python
from .rover import ScoutRover
```

From 04_package/

```
python -m expedition
```

Why this launch works

-m selects the package entry and supplies the package context.
A direct file path can leave a relative import without that context.

Read the complete explanation 讲义

Reference · Diagnose an Import

Trace **the name, the source and the initialization order**.

Symptom	Check first	Useful action
Wrong attribute	module.__file__ when available	Inspect same-named local files
Old imported value	from binding versus module attribute	Follow rebinding and shared objects
Partially initialized module	Circular access before a name exists	Separate responsibilities or delay access

Read the complete explanation 讲义

Further reading: circular imports and reload

Reference · Environment Commands

Create the environment, select its Python, then install what is needed.

macOS / Linux

```
python -m venv .venv
source .venv/bin/activate
python -m pip --version
```

Windows PowerShell

```
python -m venv .venv
.venv\Scripts\Activate.ps1
python -m pip --version
```

After activation

python -m pip install PACKAGE

PACKAGE is a placeholder; install only the dependency you actually need.

Read the complete explanation 讲义

Reference · Compilation and Module Caches

A fresh process can reuse compiled code and **still execute module initialization**.

Process cache · sys.modules

Initialized module objects
Repeated normal import can reuse them

Disk cache · __pycache__

Version-tagged compiled code
Can avoid some recompilation

Read the complete explanation 讲义

CPython overview

```
Source
   ↓ compile if needed
Code / bytecode
   ↓ execute
Module namespace
```