Advanced Inheritance

Table of contents

Outline . . . 1
A useful trick for A2: getting the class name of an object . . . 2
Abstract base classes . . . 2
Two parents with the same method name . . . 4
Single inheritance . . . 4
Multilevel inheritance . . . 6
Hierarchical inheritance . . . 6
Multiple inheritance . . . 8
`super()` and MRO . . . 12
Summary . . . 13

Lecture 10B. See csse1001[1] for course logistics, python-inheritance[2] for the full reference, now extended with abstract base classes and MRO, and 2025-10-07-uml[3] for the UML notation the diagrams below use.

Outline

- A useful trick for Assignment 2: `type(c).__name__`
- Abstract base classes (and, optionally, the `abc` module)
- Method Resolution Order (MRO), and the four inheritance models
- The diamond problem, and C3 linearisation
- `super()` and MRO

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-inheritance.pdf
[3] courses/csse1001/2025-10-07-uml.pdf

A useful trick for A2: getting the class name of an object

```
class Car():
    pass

c = Car()

print(type(c))
print(type(c).__name__) # e.g., can be used in __repr__
```

`type(c).__name__` returns the class name. The two `print`s give `<class '__main__.Car'>` and `Car`.

So `type(c).__name__` is the class name as a plain string. It is handy inside a `__repr__` (see python-dunder-methods[4]), which then stays correct even if the class is renamed or subclassed. The equivalent spelling from inside a method is `self.__class__.__name__`, used in the `abc` example below.

Abstract base classes

It is common to have an **abstract base class** that doesn't have concrete methods/attributes, but *enforces contracts* in its children classes. You're not meant to instantiate objects from these abstract classes directly.

Example: an abstract `Shape` class has an `area()` method without a concrete implementation. Every (concrete) child class of `Shape` **must provide** a concrete implementation of `area()`. Abstract base classes can sometimes have concrete implementations for *some* of their methods too (especially if those are meant to be used as-is by child classes).

```
import math

# Abstract class -- defines the interfaces (child classes must implement these methods)
class Shape:
    def area(self) -> float:
        raise NotImplementedError("Subclasses must implement area()")

    def perimeter(self) -> float:
        raise NotImplementedError("Subclasses must implement perimeter()")

class Circle(Shape):
    def __init__(self, r: float):
```

[4] courses/csse1001/python-dunder-methods.pdf

```
        self.r = r
    def area(self) -> float:
        return math.pi * self.r * self.r
    def perimeter(self) -> float:
        return 2 * math.pi * self.r

class Rectangle(Shape):
    def __init__(self, w: float, h: float):
        self.w, self.h = w, h
    def area(self) -> float:
        return self.w * self.h
    def perimeter(self) -> float:
        return 2 * (self.w + self.h)
```

The abstract class is an ordinary class whose methods carry a full signature (including the `-> float` return annotation) but no body, only a `raise`. That signature is the contract; the `raise` is what makes breaking it noisy. The `Shape` version above is a plain-Python convention, not an enforced restriction:

> `Shape()` on its own raises nothing at all, because `Shape` is still just a regular class. The `NotImplementedError` only fires later, on `Shape().area()`. Nothing actually stops you from instantiating `Shape`; only from usefully calling its unimplemented methods.

Optional: enforcing this properly with `abc`

This part is optional content and non-examinable. The standard-library `abc` module *does* enforce the contract at instantiation time:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self) -> float:
        ...
    @abstractmethod
    def perimeter(self) -> float:
        ...
    # (optional) concrete helper: allowed in an abstract class
    def describe(self) -> str:
        return f"{self.__class__.__name__}"
```

With this version, `Shape()` itself raises a `TypeError` at the point of instantiation. The abstract methods are enforced immediately, rather than only failing later when called. The exact wording of the message varies by Python version. `describe()` is the kind of concrete method an abstract base class can have: a real body in an otherwise abstract class, inherited as-is by any concrete subclass.

Two parents with the same method name

> If a class inherits from two parents, and both parents have a method with the same name, which one does Python use?

Method Resolution Order (MRO)

MRO stands for **Method Resolution Order** — it defines the order in which Python looks through classes to find a method or attribute when it's called on an object. It determines which method gets called when there are multiple implementations, and is stored in `cls.__mro__`.

Python supports several inheritance models. One running example covers all four: classes `A`, `B`, `C`, `D`, `E`.

Single inheritance

A class inherits from a single parent class. This listing is from the course notes. `A` inherits from `object`, the universal class:

```
>>> class A(object):
...     def __init__(self, x):
...         self.x = x
...     def f(self):
...         return self.x
...     def g(self):
...         return 2 * self.x
...     def fg(self):
...         return self.f() - self.g()

>>> a = A(3)
>>> a.x
3
>>> a.f()
3
```

```
>>> a.g()
6
>>> a.fg()
-3
```

`B` overrides exactly one method. First, `B` in a **flattened** form: every method `B` responds to, written out, with a comment saying where each one actually comes from:

```
>>> class B(A):
...     def __init__(self, x):              # inherit from A
...         self.x = x
...
...     def f(self):                        # inherit from A
...         return self.x
...
...     def g(self):                        # override
...         return self.x ** 2
...
...     def fg(self):                       # inherit from A
...         return self.f() - self.g()
```

That is the *effective* class. The code you actually write is only the one line that differs:

```
>>> class B(A):                             # inherit from A
...     def g(self):                        # override
...         return self.x ** 2
```

```
>>> b = B(7)
>>> b.x
7
>>> b.f()    # inherited from A
7
>>> b.g()    # overridden
49
>>> b.fg()   # inherited from A, but uses B's overridden g()
-42
```

`b.fg()` is the one to study: `fg` is `A`'s code, unchanged, but the `self.g()` inside it dispatches on the *runtime* class, so it finds `B.g`. `7 - 49 = -42`.

Multilevel inheritance

A class inherits from a child class, which in turn inherits from another parent class. It forms a linear chain of inheritance, creating a **parent → child → grandchild** relationship.

Flattened view of `C`, with the same kind of comments:

```
>>> class C(B):
...     def __init__(self, x, y):           # extend from B from A
...         super().__init__(x)
...         self.y = y
...
...     def f(self):                        # inherit from B from A
...         return self.x
...
...     def g(self):                        # inherit from B
...         return self.x ** 2
...
...     def fg(self):                       # extend from B from A
...         return super().fg() * self.y
```

There is a third category here. `B` only ever *inherited* or *overrode*; `C` **extends** `__init__` and `fg`: it calls up to the parent with `super()` and then does more. For multilevel inheritance the MRO is simple: it just follows the chain from child to parent to grandparent. Here `super()` **is** `B`.

```
>>> c = C(3, 5)
>>> c.x
3
>>> c.y
5
>>> c.f()    # inherited from B, from A
3
>>> c.g()    # inherited from B (overridden there)
9
>>> c.fg()   # extends B's fg: (3 - 9) * 5
-30
```

Hierarchical inheritance

Multiple child classes inherit from a single parent class. `D` is `A`'s second child, and it overrides a different method from the one `B` overrode:

```
>>> class D(A):                              # inherit from A
...     def __init__(self, x):               # inherit from A
...         self.x = x
...
...     def f(self):                         # override
...         return -2 * self.g()
...
...     def g(self):                         # inherit from A
...         return 2 * self.x
...
...     def fg(self):                        # inherit from A
...         return self.f() - self.g()
```

And, as with `B`, the code actually written is just the one method that differs:

```
>>> class D(A):                              # inherit from A
...     def f(self):                         # override
...         return -2 * self.g()


>>> d = D(3)
>>> d.x
3
>>> d.f()   # overridden: -2 * g() = -2 * 6
-12
>>> d.g()   # inherited from A
6
>>> d.fg()  # inherited from A, uses D's overridden f(): -12 - 6
-18
```

Three of the four models now fit in a single UML diagram:

Who defines what

The four listings as one table. Every `B`, `C` and `D` cell restates the comment on that method in the flattened view; `A` **defines** all four itself:

Method	`A`	`B`	`C`	`D`
`__init__`	**defines**	inherits	**extends** (`super()`)	inherits
`f`	**defines**	inherits	inherits	**overrides**

Method	A	B	C	D
g	**defines**	**overrides**	inherits (from B)	inherits
fg	**defines**	inherits	**extends** (super())	inherits

Multiple inheritance

A class inherits from *multiple* parent classes:

```
>>> class E(B, D):   # inherit from B and D
...     pass
```

B and D sit above E. Add A back in above them and the shape gives the problem its name.

The diamond problem

- E inherits from both B and D.
- B and D both inherit from A.
- Which version of A's methods should E use?
- Python resolves this with the **MRO C3 linearisation**.

C3 linearisation

Three rules:

- Child classes are checked before parents.
- Parents are checked in the order they are listed in the class definition.
- If a class appears multiple times in the MRO, only the *last* occurrence is kept.

Two ways to inspect the result:

```
>>> E.mro()                          # Method 1: mro() method
[<class '__main__.E'>, <class '__main__.B'>,
 <class '__main__.D'>, <class '__main__.A'>,
 <class 'object'>]

>>> for cls in E.__mro__:            # Method 2: more readable format
...     print(cls.__name__)
E
```

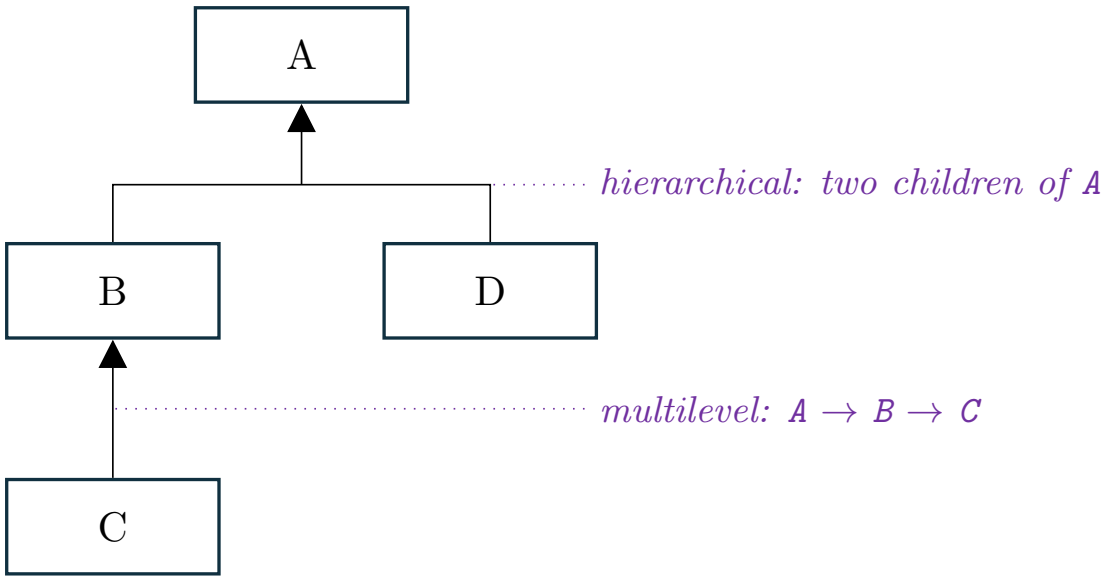

Figure 1: The running example so far: navy single-compartment boxes and the course's filled inheritance arrowhead. Every edge on its own is single inheritance; A having two children is hierarchical; the A-B-C chain is multilevel. The two purple labels name the hierarchical and multilevel models.

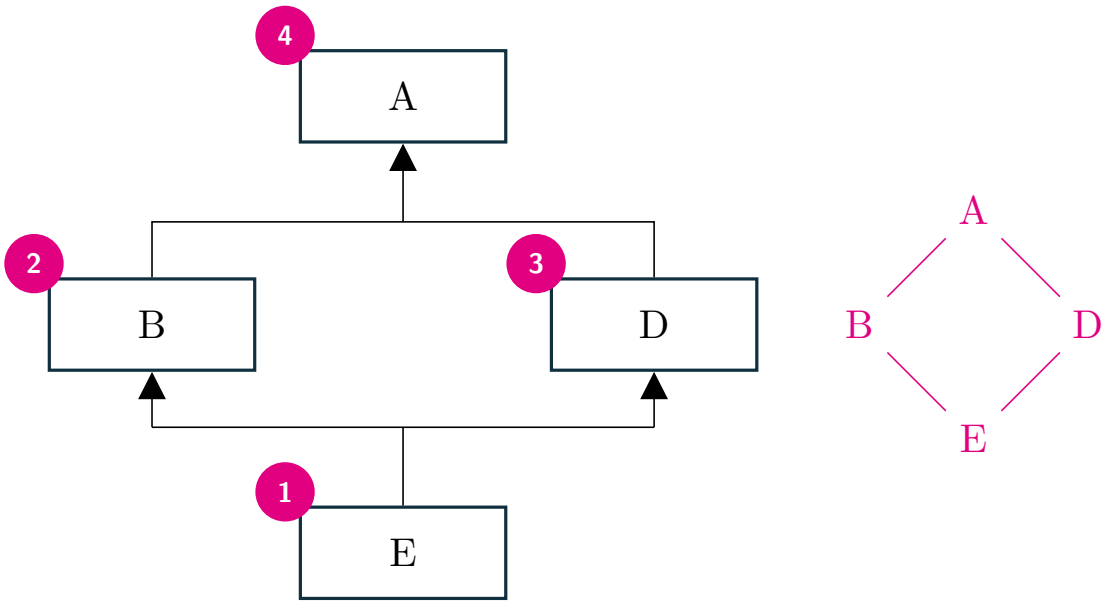

Figure 2: The diamond. The small magenta sketch beside it shows where the name comes from. The magenta numbered badges are E's method resolution order, as given by `E.__mro__`, which continues to `object` after A.

```
B
D
A
object
```

`E` defines nothing at all, so every call walks the MRO from the front:

```
>>> e = E(3)   # which method will be called?
>>> e.x
3
>>> e.f()
-18
>>> e.g()
9
>>> e.fg()
-27
```

Call	MRO walk	Resolves to	Value
`e.x`	`E`, `B`, `D` — none of them defines `__init__`; `A` does	`A.__init__` → `self.x = x`	`3`
`e.f()`	`E` no, `B` no, `D` **yes**	`D.f` → `-2 * self.g()`	`-18`
`e.g()`	`E` no, `B` **yes**	`B.g` → `self.x ** 2`	`9`
`e.fg()`	`E` no, `B` no, `D` no, `A` **yes**	`A.fg` → `self.f() - self.g()`	`-18 - 9 = -27`

Python resolves each *name* independently by walking the MRO. It does not pick one winning parent and take everything from it. `e.f()` lands in `D` and `e.g()` lands in `B`, and then `A`'s untouched `fg` calls both of them back through `self`.

A larger example, showing the general C3 rule in action:

```
>>> class A: pass
... class B: pass
... class C(A): pass
... class D(A, B): pass
... class E(C, D, B): pass
... print([cls.__name__ for cls in E.__mro__])
['E', 'C', 'D', 'A', 'B', 'object']
```

`A` is reachable through both `C` and `D`, and `B` both directly from `E` and through `D`. Applying the three rules — children before parents, bases in listed order, keep only the *last* occurrence of a repeat — gives `E`, then `C`, then `D`, then `A` (`D` lists `A` before `B`), then `B`, then `object`. The example exercises multiple inheritance.

`super()` and MRO

The `super()` function follows the *MRO*, not just the immediate parent:

```
>>> class A:
...     def ping(self):
...         print("A")
... class B(A):
...     def ping(self):
...         print("B")
...         super().ping()
... class C(A):
...     def ping(self):
...         print("C")
...         super().ping()
... class D(B, C):
...     def ping(self):
...         print("D")
...         super().ping()

>>> D().ping()
D
B
C
A
```

`D`'s MRO is `[D, B, C, A, object]`. When `B.ping()` calls `super().ping()`, it does **not** jump straight to `A` — `B`'s statically-declared parent — it calls the *next class in the actual runtime MRO of the instance*, which is `C`. This is what makes cooperative multiple inheritance work: each class's `super()` call advances one step through the shared MRO, regardless of what its own declared parent is.

> If a class in the chain doesn't call `super()` at all, the chain of calls simply stops there — the MRO itself is unaffected (it's purely a function of the inheritance structure), but fewer `ping()` implementations actually get executed. For example, if `B.ping()` omits its `super().ping()` call, `D().ping()` only prints `D` and `B` — `C` and `A` are never reached, even though they're still part of `D`'s MRO.

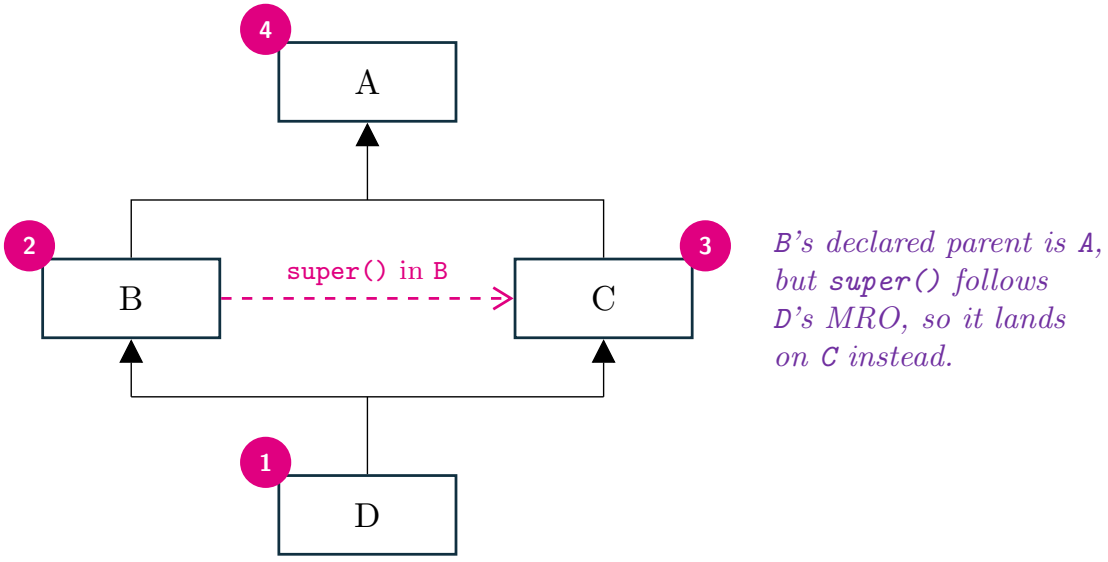

Figure 3: Why super() in B lands on C, for the D(B, C) hierarchy. The class boxes and filled inheritance arrows are the course's UML notation. The magenta badges give the print order D B C A, and the dashed magenta arrow is the delegation made by super() in B.

Summary

When a class inherits from multiple parents, it's possible for more than one parent to define the same method or attribute. To avoid confusion and ensure consistency, Python uses a rule called **Method Resolution Order (MRO)** to determine the order in which classes are searched. MRO follows a well-defined path based on class hierarchy and inheritance order, ensuring that each method or attribute is found in a predictable and logical way. This is especially important in complex inheritance situations like the diamond pattern.

Next: design patterns and MVC (2025-10-14-model-view-controller[5], Week 11).

[5] courses/csse1001/2025-10-14-model-view-controller.pdf