Advanced Inheritance

Table of contents

Today's outline 1
A useful trick for A2: getting the class name of an object 1
Abstract base classes 2
Advanced inheritance 3
`super()` and MRO 8
Summary 9

See csse1001[1] for course logistics — this note covers Lecture 10B's technical content. See python-inheritance[2] for the full reference, now extended with abstract base classes and MRO.

Today's outline

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

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

```
class Car():
    pass

c = Car()
print(type(c))          # <class '__main__.Car'>
print(type(c).__name__)  # 'Car' -- e.g. can be used in __repr__
```

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-inheritance.pdf

`type(c).__name__` returns the class name as a plain string — handy inside a `__repr__` (see python-dunder-methods[3]) so it stays correct even if the class is renamed or subclassed.

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):
        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)
```

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

```
s = Shape()   # BAD -- you're not meant to create an object from this abstract base class
s.area()      # ERROR
```

> `Shape()` itself doesn't raise an error here — it's still just a regular class. The error only happens on `s.area()`, which raises `NotImplementedError`. This is a plain-Python convention, not an enforced restriction: nothing actually stops you from instantiating `Shape`, only from usefully calling its unimplemented methods.

Optional: enforcing this properly with `abc`

Using the standard-library `abc` module *does* enforce this 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 `TypeError: Can't instantiate abstract class Shape with abstract methods area, perimeter` — the abstract methods are enforced immediately, rather than only failing later when called.

Advanced inheritance

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:

Single inheritance

A class inherits from a single parent class:

```
>>> class A(object):   # 'object' is the universal class
...     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

>>> class B(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
```

Multilevel inheritance

A class inherits from a child class, which in turn inherits from another parent class — forming a linear parent → child → grandchild chain:

```
>>> class C(B):
...     def __init__(self, x, y):   # extends B's (and A's) __init__
...         super().__init__(x)
...         self.y = y
...     def fg(self):                # extends B's (and A's) fg
...         return super().fg() * self.y

>>> 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
```

For multilevel inheritance, the MRO is simple — it just follows the chain from child to parent to grandparent, etc.

Hierarchical inheritance

Multiple child classes inherit from a single parent class:

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

>>> d = D(3)
>>> d.x
3
>>> d.f()   # overridden: -2 * g()
-12
```

```
>>> d.g()   # inherited from A
6
>>> d.fg()  # inherited from A, uses D's overridden f()
-18
```

The combined UML diagram for `A`, `B`, `C`, `D` above:

```
    A
   /  \
  B    D
  |
  C
```

Multiple inheritance

A class inherits from *multiple* parent classes:

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

```
 B   D
  \ /
   E
```

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?

```
    A
   / \
  B   D
   \ /
    E
```

Python resolves this with **MRO C3 linearisation**:

- 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.

```
>>> E.mro()
[<class '__main__.E'>, <class '__main__.B'>,
 <class '__main__.D'>, <class '__main__.A'>,
 <class 'object'>]

>>> for cls in E.__mro__:
...     print(cls.__name__)
E
B
D
A
object


>>> e = E(3)
>>> e.x
3
>>> e.f()    # E has none; B has none of its own; D's f() is used
-18
>>> e.g()    # B's overridden g() is used (B comes before D in the MRO)
9
>>> e.fg()   # A's fg(): self.f() - self.g() = -18 - 9
-27
```

Even though `B` doesn't define its own `f()`, and `D` doesn't define its own `g()`, Python resolves each *name* independently by walking the MRO — `e.f()` finds `D`'s `f()` (since `B` has none of its own), while `e.g()` finds `B`'s `g()` (since `B` comes before `D` in the MRO).

A larger example, showing the general C3 rule (child before parents, parents in listed order, keep only the last occurrence of a repeated class):

```
>>> 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']
```

`super()` and MRO

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

```
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 doesn't 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.

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 (Week 11).